LALPC: Exploiting Parallelism from FPGAs Using C Language

Lucas F. Porto[1], Marcio M. Fernandes[1], Vanderlei Bonato[2], Ricardo Menotti[1]

[1] Department of Computer Science
UFSCAR - Federal University of São Carlos
São Carlos - SP, Brazil
[2] Institute of Mathematical and Computer Sciences
USP - University of São Paulo
São Carlos - SP, Brazil

E-mail: `menotti@dc.ufscar.br`

Abstract. This paper presents LALPC, a prototype high-level synthesis tool, specialized in hardware generation for loop-intensive code segments. As demonstrated in a previous work, the underlying hardware components target by LALPC are highly specialized for loop pipeline execution, resulting in efficient implementations, both in terms of performance and resources usage (silicon area). LALPC extends the functionality of a previous tool by using a subset of the C language as input code to describe computations, improving the usability and potential acceptance of the technique among developers. LALPC also enhances parallelism exploitation by applying loop unrolling, and providing support for automatic generation and scheduling of parallel memory accesses. The *combination* of using the C language to automate the process of hardware design, with an efficient underlying scheme to support loop pipelining, constitutes the main goal and contribution of the work described in this paper. Experimental results have shown the effectiveness of those techniques to enhance performance, and also exemplifies how some of the LALPC compiler features may support performance-resources trade-off analysis tasks.

1. Introduction

Nowadays, reconfigurable computing, in particular modern FPGAs (*Field-Programmable Gate Arrays*), can be considered a good platform for the development of highly parallel systems with low power consumption. Many hybrid architectures adopt software to execute large code sections, and hardware accelerators to run performance critical sections of a given application. Due to software code reuse, this approach is able to increase performance while improving productivity in the development process [1].

In embedded systems, the hardware/software combination is usually tightly coupled, either using a reconfigurable soft-core processor in the device itself, or via devices and kits that combine FPGAs and a processor. In this context, it is worth mentioning some contemporary solutions such as Xilinx Zynq FPGA family [2], which includes ARM cores hardwired, and Altera/Intel DE2i-150 kit [3], which interconnects an Atom processor with a Cyclone IV FPGA through a PCIe bus. Several recent studies report the use of FPGAs to accelerate applications, such as [4], and many others presented in [5].

Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

Despite the potential for high performance gains and power savings, reconfigurable computing is still considered by many designers as a hardware solution requiring long development times. This limitation is of particular concern for the adoption of hybrid architectures relying on FPGA accelerators. HLS (*High-Level Synthesis*) tools – used to convert more abstract descriptions directly into circuits – have been proposed since the 90s, in an attempt to facilitate hardware development, bringing it closer to software developers. After more than 20 years of steady progress, more mature tools have emerged, and those have become almost a necessity for the following reasons: i) need for rapid design space exploration from functional specifications; ii) high complexity of current devices, requiring a higher level of abstraction for large projects; iii) need for improved productivity by means of components reusing; iv) need for accurate system-level verification; v) increasing adoption of accelerators and heterogeneous SoCs (*System-on-Chips*) [6].

Several challenges face the design of a HLS tool. Besides the goal of delivering RTL (*Register-Transfer Level*) circuits as efficient as possible regarding aspects such as area, performance, and energy consumption, the input language is determinant for the tool adoption and effective use. One of the approaches presented in Section 2 is LALP (*Language for Aggressive Loop Pipelining*), a scheme aimed to generate hardware accelerators targeting the optimization of software loops. The technique relies on a special purpose language, and underlying hardware model specially designed to optimize the execution of pipelined loops [7].

Experimental results using LALP techniques have shown promising results in terms of time and area, which has motivated further developments to improve the way computations are specified by a system's designer: instead of using the special purpose language, the framework has been extended to use the C Language, allowing for easier translation of software algorithms into hardware structures. The use of C language is a desirables feature as most algorithms of interest for embedded systems have well tested reference implementations described in that language. The LALPC (*C-based Language for Aggressive Loop Pipelining*) compiler infrastructure presented in this paper extends the functionality of LALP not only in terms of the input language, but also in the strategies used to increase parallelism exploitation [8]. Besides loop pipelining, automatic loop unrolling and parallel memory access are extensively used whenever possible, and thus improving overall performance results. By doing so, LALPC improves the tool usability and acceptance through the use of C language, while still retaining the area and time advantages of the underlying hardware scheme supporting loop pipelining. The *combination* of those two features constitutes the main contribution of this work, which has shown promising results when compared against some current similar state-of-the-art tools.

The remaining sections of this paper are organized as follows: Section 2 presents some related works. In Section 3 the LALPC compiler infrastructure and its main features are described, followed by some experimental results in Section 4. Section 5 brings some final remarks regarding the work presented in this paper.

2. Related Work

Several high-level synthesis tools have been proposed recently using various approaches, which shows the growing interest in automatic generation of hardware for reconfigurable devices [9]. Among others, some notable academic works are Spark [10], Haydn-C [11], ROCCC [12], and MATISSE [13]. Commercial tools have also matured, and a number of them are currently offered, such as Altera SDK for OpenCL, Xilinx Vivado, Calypto Catapult, Synopsys Synphony C, Cadence C-to-Silicon, ImpulseC and Cynthesizer. Metamodeling has also being exploited for system level design [14], where hardware and software are represented in a common environment. Such approach allows a hardware synthesis tool to use several information from the entire system in order to meet requirements.

An interesting alternative is ReflectC, a high-level synthesis tool with a focus on improving flexibility during the design process [15]. ReflectC can be used in conjunction with LARA [16], a special purpose aspect-oriented programming language. LARA has been designed for code instrumentation, controlling the application of code transformations and compiler optimizations. The language allows developers to capture non-functional requirements from applications in a structured way, while still keeping the original application source-code. By doing so, it enables effective application and control of the different tools available in the high-level synthesis toolchain, taking into account domain and target-specific aspects. This strategy improves the efficiency and coverage of the design-space exploration process, and ultimately may expose additional opportunities for hardware-software partitioning.

LegUp [17] is an open source project that has received considerable attention from the scientific community. The tool focus is on high-level synthesis using the C language as input. The C code is automatically compiled, and corresponding hardware structures are generated, described in Verilog HDL (*Hardware Description Language*). LegUp uses as front-end the LLVM compiler [18], which also performs the initial steps of analysis and optimization for further synthesis. LegUp implements a new back-end in order to target a hardware platform [17]. The tool can synthesize a substantial part of the C language, supporting various structure types, such as multidimensional arrays, global variables, pointers, and even floating-point values. The hardware generated by the LegUp tool has comparable quality to those generated by some available commercial tools for high-level synthesis [19]. In addition, it allows the integration of various accelerators with programs described in OpenMP and Pthreads [20].

The LALP compiler adopts a focused approach to high-level synthesis, aiming to optimize algorithms relying on time critical loops [7]. The basis of this approach is called ALP (*Aggressive Loop Pipelining*), a hardware scheme which adapts some of the ideas proposed in [21] for execution in FPGA platforms. However, instead of a data-driven scheme, the ALP technique attempts to achieve the maximum throughput by using counters to furnish the iteration space in loops, and shift-registers to synchronize operations [22]. By doing so, it avoids the need of a centralized control based on finite state machines, a *key difference* from traditional approaches to loop pipelining. In the ALP scheme, operations in the loop body are executed at clock cycles according to the paths taken during execution, and optimal loop pipelining can be achieved, without performance losses due to an FSM controller.

The LALP tool allows design space exploration through marking directives in the code. These directives are used to control the exact clock cycle of operations, generating different architectures for the same input code. Another important feature is that the LALP compiler uses a specific language for describing the algorithm. The LALP language has a higher level of abstraction when compared to VHDL and Verilog languages. The results obtained for this domain may be very efficient, as demonstrated in [23], however, the use of a *special purpose language* limits its use since it implies a learning curve for any developer wishing to use it.

Considering that the LALPC tool presented in this paper was built upon ALP, a specialized hardware scheme offering time and area gains for loop execution over some other existing tools [22], it can be argued that LALPC keeps this advantage over similar HLS tools. The support for C language input extends LALPC applicability, and so its chances of outperforming existing HLS tools for loop optimizations.

3. LALPC Compiler

The LALPC compiler presented in this paper extends LALP to support the C language as input, and also employs new strategies for parallelism exploitation. As already pointed out, one of the main changes in relation to the LALP compiler is the use of C language as input for describing computations. This has been accomplished by using the ROSE compiler [24] as part of the overall LALPC compilation flow (Figure 1).

As a prototype implementation, LALPC allows for a limited subset of C language, although sufficient for testing purposes with representative application codes. In addition to sequential constructs, it is possible to use the following constructions of the C language: regular (for) loops, block and ternary conditional checks, arithmetic and logic operations, accumulators, arrays, bit shifts. Some notable limitations include float data types, function calls, structs, pointers, multi-dimensional arrays, and recursion. Such limitations are due to the current capabilities of the LALPC back-end, some of them usually found in other HLS tools (e.g. float data types, function calls).

The LALPC compiler targets the same VHDL components library used in LALP [25]. This hardware library is optimized for *loop pipelining* execution, a performance oriented feature at the core of the system. The generated VHDL code can be synthesized both in Altera and Xilinx tools.

The compiler allows using markup directives to guide the tool during the hardware generation process. These markup directives are described as *pragmas* inserted in the code, which are them handled by the compiler. By varying some parameters through the use of pragmas, it is possible to generate different architectures with specific features for the same input source code. With pragmas, the programmer can apply hardware customizations based on knowledge from a high abstraction level of implementation details. Pragmas also controls the use of other features aiming to improve parallelism exploitation, in particular *loop unrolling*, and the use of *multi-port memories* for simultaneous array accesses. The use of pragmas in LALPC compiler is described in details in Section 3.2.

The previous LALP tool required the programmer to use directives to control the timing of operations. This manual synchronization ensures that the scheduling process will always run properly. Although the LALPC compiler still supports the use of manual synchronization (by using pragmas), it is no longer a requirement. By relying on the more robust analysis passes and intermediate representation provided by the ROSE compiler, it is possible for the framework to infer valid synchronized sequences of operations. All tests conducted in this study showed the correctness of this automatic approach, however, it does not implies on a formal compiler validation at this stage.

3.1. Compiler Structure

The LALPC instantiates parts of the ROSE compiler to validate the input code in C language and to apply some optimization techniques. After this step, an intermediate representation, built in the form of an AST (Abstract Syntax Tree) with the intermediate representation of the code is obtained. This representation is used to generate the necessary hardware components, and to define the data and control connections among them.

The ROSE compiler is capable of performing a number of analyses and optimizations in the input code. Of particular interest is the code representation in Static Single Assignment (SSA) form, making unique every assignment to a variable. This optimization eases the hardware generation process, a step manually performed for the previous compiler.

Some pragmas are handled directly by the ROSE compiler front-end, which are reflected in the AST representation (e.g. loop unrolling). Others are dealt with by the hardware-oriented back-end (e.g. operations bit-width). After creating the signals among components, the compiler performs the scheduling and balancing steps [26]. Finally, the VHDL compiler generates files with the components and their connections. Figure 1 shows the structure of the compiler, which also outputs auxiliary internal compiler representations for a visualization tool (Graphviz).

3.2. Pragmas

High-level synthesis tools (both research and commercially oriented) usually have limitations arising from differences between the paradigms of software and hardware development. Software programming languages aim to map a given problem in a well-defined general-purpose

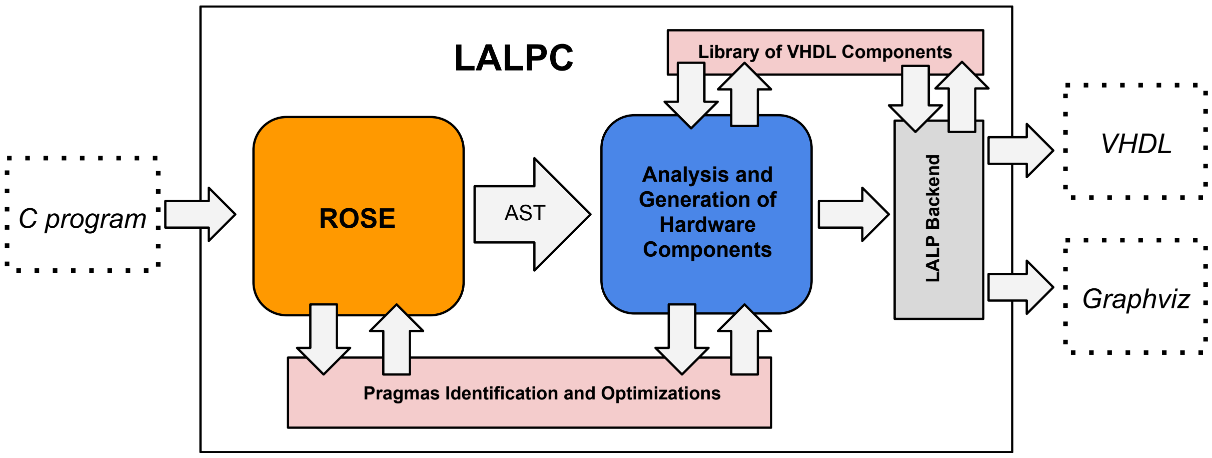

LALPC
Library of VHDL Components
C program
ROSE
AST
Analysis and Generation of Hardware Components
LALP Backend
VHDL
Graphviz
Pragmas Identification and Optimizations

Figure 1: The LALPC Compiler Structure

architecture. The construction of a custom hardware architecture for a given problem may take into consideration different parameters, since there is no fixed bounding to previously defined functional units, memory hierarchy and so on. These differences may lead the synthesis tool to generate architectures that require a large amount of hardware resources. Code restructuring and/or fine tuning of hardware generation parameters is often a strategy for mitigating undesirable effects.

Programming directives via pragmas is intended to allow the programmer to define some specific characteristics during the build process. The use of pragmas modifies the hardware generated by the compiler, which can result in different architectures. For the same input code, not only different performance levels may be obtained, but significant variations in resources usage and power consumption in the adopted reconfigurable device. The use of pragmas is also justified by the existing features and limitations in C when used for hardware specification.

To represent the different paradigms between the software described in C language and hardware architectures, consider, for example, the need to specify a function with multiple outputs. In the LALPC compiler it is possible to set pragmas that indicates which variables may have output pins in their respective registers.

The pragmas currently treated by the LALPC compiler are described below:

- `#pragma alp bit` – defines a boolean (single bit) register, since the C language does not have a data type for this. This pragma is useful to define control signals, preventing wider variables being used for this purpose.
- `#pragma alp data_width` – defines the bit width used by the VHDL components. This program can be used to increase or decrease the field of representation of numbers according to the need, providing a space-saving mechanism. The VHDL component library is fully parameterized in terms of bit width, the compiler can then instantiate components with the width specified in this pragma.
- `#pragma alp delay` – controls clock cycles of hardware components, allowing manual synchronization of operations. This feature can be used to improve the circuit's operating frequency (e.g. putting additional registers), which enables the retiming in synthesis tool.
- `#pragma alp multiport` – checks for multiple memory accesses, and generates custom multiport RAM modules. This feature is explained in more detail below.
- `#pragma alp out` – creates output pins in VHDL components, typically used to facilitate data analysis and the generation of blocks with multiple outputs. The execution of a

function in software is temporal, resulting in a value at the end of all steps. In hardware – with spatial execution in pipeline – a block can simultaneously calculate several iterations. This pragma can then be used to extract intermediate values of the computation, which can be useful in other parts of the algorithm.

- `#pragma alp unroll` – used to apply the loop unrolling optimization, resulting in performance gains for application codes based on loop-critical sections, as can be seen below. The handling of this pragma is delegated to the ROSE compiler, invoking a method that works directly in its intermediate representation.

3.3. Parallelism Exploitation

The *multiport* and *unroll pragmas* are used to apply optimization techniques to speed up the execution speed of applications. However, its use may also affects the amount of resources allocated in the reconfigurable device, as can be seen below. For this reason, a level of design space exploration may be required to balance performance and resources utilization.

3.3.1. ***#pragma alp multiport*** Most of today's FPGAs have dual port memories internally. This pragma generates a RAM component with multiple ports, allowing simultaneous read access. Write parallel access are limited to a factor of two. This improves the final performance of the application in many cases: serial array access only constrained by memory ports can be overcame by the use of multiport memory banks, allowing for parallel access. The generation of these memories is delegated to the synthesis tool and is beyond the scope of this paper. To have more control over the process that could be treated with techniques such as those described in the literature [27, 28, 29].

3.3.2. ***#pragma alp unroll*** The use of the `#pragma alp unroll` instructs the compiler to analyse the operations within the loop, trying to identify dependencies between them. If dependencies permit, the compiler duplicates internal components of the loop, enabling more than one iteration to be executed in parallel. The result is a significant gain in the application processing when compared with the sequential version. On the other hand, the process of replicating internal operations increases the amount of used resources in the reconfigurable device, a growth proportional to the number of operations contained in the loop.

As an example, Figure 2a aims to exemplify the use of this pragma: line 4 specifies that, if possible, an unroll factor of two should be applied to the loop body. Figure 2b shows the resulting code generated by the ROSE compiler (in high-level language, for clarity purposes). This new structure becomes the intermediate representation that will be used by the LALPC back-end, and correponding generation of hardware components.

```
1 void vecsum() {
2   int i, x[N], y[N], z[N];
3   #pragma alp multiport
4   #pragma alp unroll 2
5   for (i = 0; i < N; i++)
6     z[i] = x[i] + y[i];
7 }
```

(a) Initial code.

```
1 void vecsum() {
2   int i, x[N], y[N], z[N];
3   for (i = 0; i < N; i+=2) {
4     z[i]   = x[i]   + y[i];
5     z[i+1] = x[i+1] + y[i+1];
6   }
7 }
```

(b) Resulting code.

Figure 2: Example of *loop unrolling* provided by ROSE compiler

It can be seen that line 5 was generated from line 4, but with different access to the memories, which allows parallel processing of two iterations for each clock cycle.

Also in this example, when applying the unroll pragma the compiler generates more simultaneous accesses to the memories, being represented by the indexes “i” and “i+1” of the vectors. In this way it is possible to combine the pragma multiport for simultaneous access to this memory. Line 3 on Figure 2a contains the pragma that indicates to the LALPC compiler customize the memories when simultaneous accesses are found, regardless of reading or writing.

The combination of the two above pragmas increases significantly the performance in many applications, compared to the sequential processing. It is worth to note that the use of such features is an option when the developer needs to meet project requirements. Section 4 presents some experimental results showing the impact of using these pragmas in terms of performance and resources usage.

4. Experimental Results

Some experimental results were conducted in order to evaluate the hardware generated by the LALPC compiler. This work focused on performance and resources usage, comparing the results against the previous LALP compiler, and also against the LegUp compiler. Although all selected benchmarks rely heavily on loops, they present different characteristics in terms of size of the loop body, pattern and frequency of memory access, and data dependences. All benchmarks use 32-bit integers, unless stated. The benchmarks used for this work are:

- Accumulator algorithm [17] - 10 loop iterations;
- ADPCM encoder/decoder audio algorithms [30] - 1024 loop iterations;
- Sobel image processing algorithm [31] - 78 loop iterations;
- Dotprod, Max and Vecsum algorithms used in DSPs [32] - 2048 loop iterations;

The synthesis tool used was Quartus II 13.0.0 Web Edition (64-bit) targeting Altera EP4CGX150DF31C7 Cyclone IV GX Family reconfigurable device (FPGA). Terasic DE2i-150 board was used as a platform both for software and hardware tests. Software results were adopted as a baseline implementation, a reasonable choice when porting software implementations to hardware platforms. The software execution was performed on a Intel Atom Dual Core Processor N2600, with a 1MB cache and running at 1.6GHz. The adoption of the Atom microprocessor was due to its tight integration to the FPGA development board used for the hardware implementations presented in this work. Considering that this processor is offered as part of a commercial hardware-software hybrid solution, it can be argued that it is a representative microprocessor of a hypothetical software-only embedded system. In all tests performed for this work, only implementation with one thread of execution were employed.

Software performance figures were obtained with *perf-stat* tool running on Yocto 3.0.32 and Poky 1.3.2 reference system which is based on GCC 4.7.2. The raw data obtained for performance and hardware resources usage are showed in Table 1, and further discussed in the next subsections.

4.1. Performance

The data presented in Figure 3 compares the normalized execution times for all benchmarks running on three hardware architectures and on one microprocessor for the software implementation. As expected LALPC and LALP results are very similar, as they target the same underlying hardware components, highly specialized to loop pipelining execution. This is also the very reason why LALPC outperforms LegUp for the selected benchmarks. However, this advantage should not be expected in general implementations.

Table 1: Hardware resources usage and execution time for software platform (Intel Atom), and hardware implementations generated by LegUp, LALP and LALPC.

Benchmark	**Platform Compiler**	**Logic Elements**	**Comb**	**Reg**	**Memory (Bits)**	**Execution Time(us)**
Accumulator	Software	–	–	–	–	13.31
	LEGUP	658	564	541	640	0.44
	LALP	163	109	138	512	0.11
	LALPC	159	109	139	512	0.10
ADPCM Coder	Software	–	–	–	–	17.48
	LEGUP	1067	995	661	36128	172.10
	LALP	800	638	37	40960	79.72
	LALPC	986	673	837	41460	119.30
ADPCM Decoder	Software	–	–	–	–	16.90
	LEGUP	1049	913	683	52512	95.97
	LALP	507	388	464	41090	16.61
	LALPC	628	420	574	41248	39.00
Dotprod	Software	–	–	–	–	43.79
	LEGUP	852	696	590	131072	152,09
	LALP	95	81	65	131072	28.92
	LALPC	118	102	82	131072	28.50
Max	Software	–	–	–	–	38.55
	LegUP	294	272	234	65356	31.46
	LALP	66	47	62	65356	12.79
	LALPC	94	75	84	65356	12.89
Sobel	Software	–	–	–	–	24.06
	LEGUP	1266	1156	820	6400	4.16
	LALP	797	683	504	8269	5.67
	LALPC	1158	754	991	8435	4.40
Vecsum	Software	–	–	–	–	43.92
	LEGUP	891	788	551	196608	117.85
	LALP	101	53	67	196608	18.95
	LALPC	124	74	83	196608	17.20

Although for most benchmarks there are modest performance gains when moving from LALP to LALPC, it is worth noting that this can be accomplished with much less development effort, since algorithms can be coded using C language in LALPC, without the need of explicit synchronization parameters. In LALP, ADPCM implementations were modified to achieve greater parallelism, severely altering the order of operations in the code. In LALPC version we prefer to keep the unmodified code to make a fair comparison with LegUp.

As seen in Table 1, most hardware implementations result in clearly better performance when compared to the software platform. The noticeable exception is the ADPCM coder algorithm. That is explained by the algorithm characteristics, which have data dependences preventing the use of loop pipeline optimizations. That results in a quasi-sequential execution at a much lower clock rate in the hardware platform, when compared to the 1.6 GHz Atom processor. The obtained results for the software implementations are the mean value based on five runs for each benchmark, presented within a 95% confidence interval.

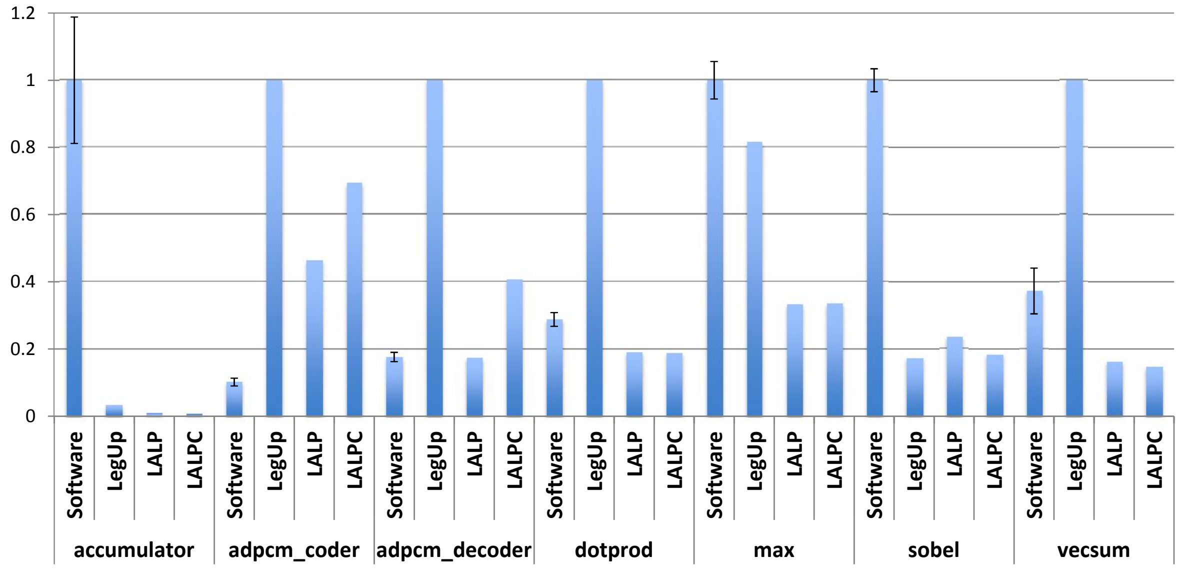


Figure 3: Comparison between normalized execution times for all implementations.

4.2. Speedup

For four benchmarks, the data in Table 2 shows the results obtained using optimization pragmas in terms of execution time and speedup, comparing all there compilers: LALPC, LALP, and LegUp. LALPC acheived an average speedup over LALP of 4.7x, and 10.6x when compared to LegUp. Those results demonstrated the effectiveness of the optimization techniques implemented in this work, for the selected benchmarks. However, the performance gain is usually obtained at the expense of additional hardware resources, as will be shown in the next sections. This observation leads to the need of trade-off analysis, shown in Sections 4.4 and 4.5.

Table 2: Speedup obtained using optimization pragmas.

Benchmark	**Time LALPC (us)**	**Time LALP (us)**	**Speedup**	**Time LegUp (us)**	**Speedup**
Accumulator	0.05	0.11	**2.2**	0.44	**8.8**
Dotprod	3.95	28.92	**7.3**	152.09	**38.5**
Sobel	0.45	5.67	**12.6**	4.16	**9.2**
Vecsum	9.73	18.95	**1.9**	117.85	**12.1**

4.3. Hardware Resources Usage

It can be observed in Table 1 that, in general, LegUp architectures requires more logic elements as it relies more heavily on Finite State Machines (FSMs), as opposed to LALPC and LALP. As for internal memories, most resources usage are similar, apart from those for the accumulator benchmark, which employs flip-flops instead of memories, in LALPC and LALP. Those two compilers always generate memory capacity (in number of words) as a power of 2, while LegUp is able to generate memories of arbitrary sizes. That can be clearly observed for the Sobel benchmark. LegUp instantiates memories using an IP core from Altera, while LALPC and LALP generates custom memories in VHDL.

As shown in Figure 4 (using a logaritmic scale) hardware implementations generated by LALPC are similar to those generated by LALP, in terms of hardware resources usage. This aspect was evaluated in terms of number of logical elements, combinational functions, logic registers, and memory bits. However, the results for both compilers are not identical, explained by the use of a new front-end and additional code transformations performed by the ROSE compiler. As already observed for the performance results, LALPC compares favourably against LegUp for the benchmarks used in this study. The most likely reason is again the specialized hardware components targeted by LALPC, an effect unlikely to appear for application codes not relying on loops.

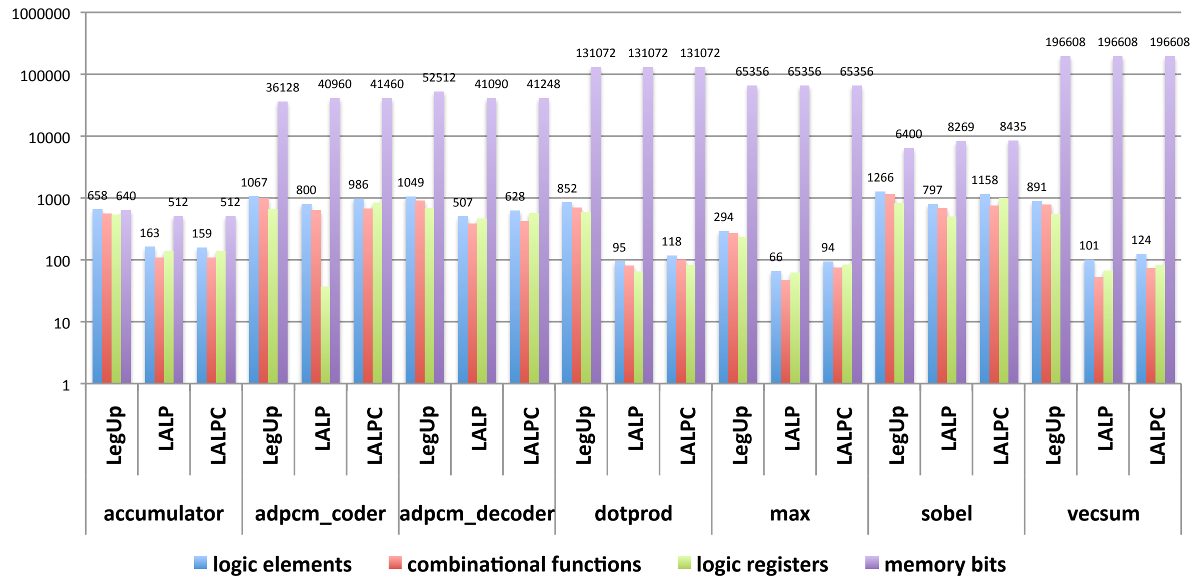


Figure 4: Hardware resources usage.

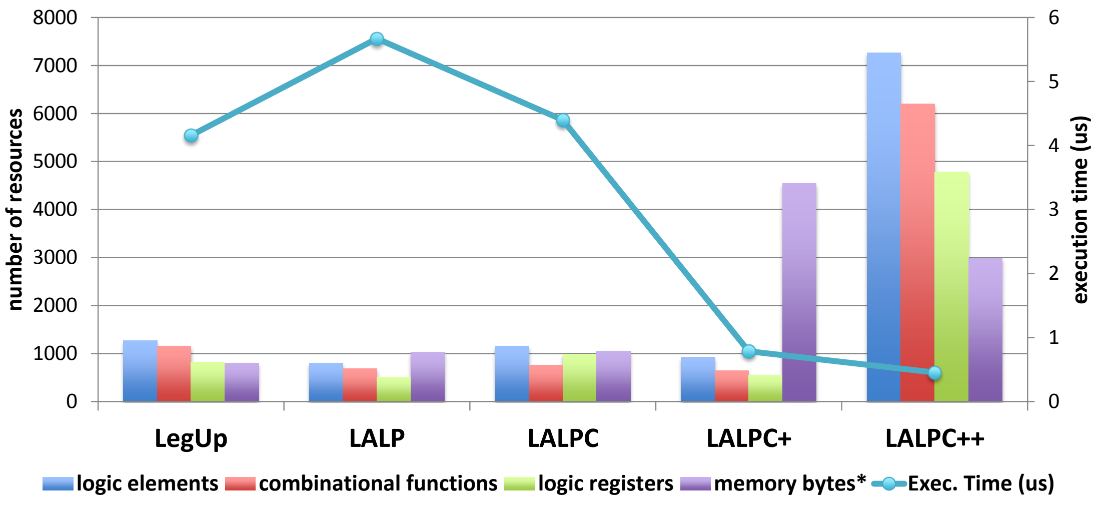
4.4. Design Space Exploration: Sobel

The use of pragmas has been already mentioned as a mechanism to help the designer in the task of fine tuning the compiler output, allowing for an easy exploration of some solution points trying to find an adequate balance between performance and hardware resources usage. In order to demonstrate the effective of this LALPC compiler feature, the Sobel benchmark was used. The Sobel benchmark was chosen due to some of its characteristics, it permits tests with the pragmas separately. One of the characteristics of interest of the algorithm is that it performs, for each loop iteration, 8 read accesses in the input vector, and a single write in the output vector after those data are processed. A number of hardware implementations were generated for the same input code, only varying the use of pragmas to control the level of optimizations employed.

In order to perform 8 read access to the RAM memory, a multiplexer is used to schedule the access to distinct addresses, making the reading step fully sequential. This process requires 8 cycles to read the pixels from memory for each iteration of the repeating loop. By using the *pragma multiport*, the compiler is able to replace the single port RAM memory module by a RAM memory with 8-read and 8-write ports. By using this customization, all read accesses are executed in a single cycle, and thus allowing a significant performance gain. However, this parallelism is achieve at high cost in terms of resources, since the actual implementation consists of 4 dual-port memories, with replicated input data.

Another optimization possible in this example is *loop unrolling*, because there are no dependencies within the repeating loop. By combining this optimization with the multiport pragma, the resulting hardware ensures a further performance gain, although at the expense of additional hardware resources. This combination doubles all existing operations within the loop kernel, generating 16 simultaneous accesses to the memory module containing the input image, and other two write accesses in the output memory. Unrolling a loop body increases the complexity of logic circuits as some of the data used in computations are moved from memory to internal registers. That causes a reduction in the number of employed memory bits, and an increase in the use of logic elements.

This performance-hardware trade-off can is graphically presented in Figure 5.



LALPC+: Multiport — LALPC++: Multiport & Unroll

Figure 5: Sobel Design Space Exploration: Resources vs Execution Time

Another evaluation easy performed by using the pragmas is the impact resulting from varying the memory and operations bit-width employed by the resulting architecture (Figure 6).

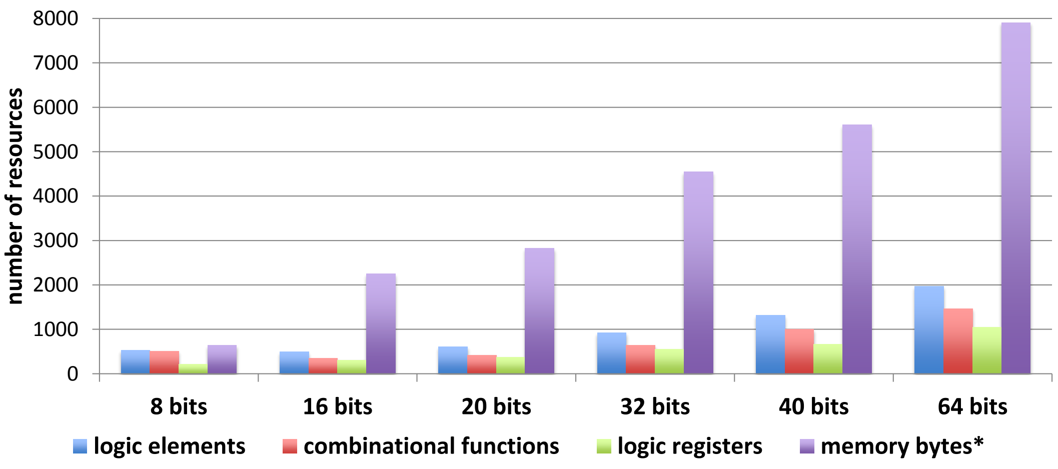


Figure 6: Sobel Design Space Exploration: Memory and Operations Bit-width

Those experiments exemplify the potential of the optimization techniques implemented in LALPC compiler. However, it is worth noticing that they also may affect the amount of resources required for the hardware synthesis, and could make it infeasible depending in case of limited resources in the reconfigurable device adopted for a given project.

4.5. Design Space Exploration: Accumulator, Vecsum, Dotprod

After completion of the initial tests for the evaluation of pragmas with the Sobel benchmark, new tests were performed with some of the other benchmarks mentioned above. However, applications that have intra-iteration loop dependences do not benefit from the unroll pragma. Because of that, some benchmarks of the list were not considered in this analysis. As shown in Figure 7, the same results pattern observe for Sobel were observed for the Accumulator, Vecsum, and Dotprod benchmarks: a steady improvement in performance resulting from the use of loop unrolling and parallel memory accesses, at the expense of additional hardware resources. Although this result is expected, it should be noticed that the LALPC compiler allows a quantitative evaluation of all design points with little modifications in the source code. The results for the accumulator benchmark show little variation in resources usage, but also little performance gains, which is due to a small number of loop iterations in the code. Execution time is halved for Vecsum, but it does not scale very well (exponential), as all data are moved to registers. Dotprod relies on resource replication but without increasing the use of memory, resulting in linear scaling for resources usage.

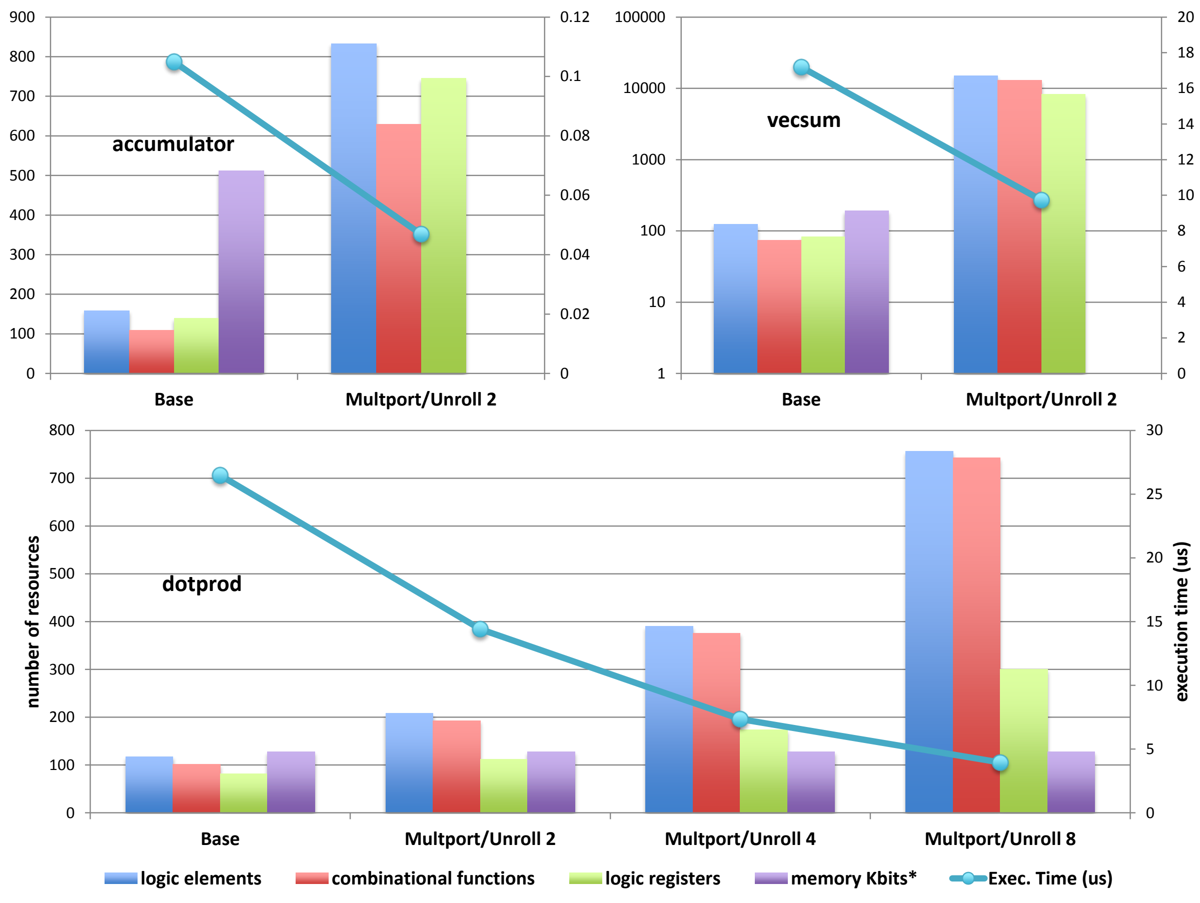


Figure 7: Accumulator, Vecsum, Dotprod Design Space Exploration: Resources vs Execution Time

5. Conclusion

This work has presented the LALPC compiler, a high-level synthesis tool, which was built as an extension of an existing tool (LALP). From the user's point of view, the main advantage in relation to the previous tool is the use of C language as input code to describe computations. LALPC also supports the use of pragmas to guide the hardware generation process, and to enable some optimizations such as loop unrolling, and parallel memory access. It has been demonstrated that those features can be useful support time-area trade-off analyses during the development process. The combination of using C language with an efficient hardware scheme supporting loop pipelining constitutes the main contribution of this work.

Although allowing only a subset of the full C language specification, the LALPC compiler was able to generate efficient implementations for some loop-oriented kernel benchmarks. Those results compare favourably against those obtained using the LegUp compiler, although we acknowledge this can be achieve only for loop intensive code segments. It is also worth noticing that LegUp supports a broader C language subset, which obviously increases its applicability when compared to LALPC. Nevertheless, the results obtained in this work suggests that the specialized techniques employed by LALPC have the potential to be further improved, or incorporated as a specialized feature targeting loops on other existing HLS tools.

References

[1] Liao C, Quinlan D J, Willcock J J and Panas T 2010 *International Journal of Parallel Programming* **38** 361–378
[2] Santarini M 2011 *Xcell J* **75** 8–13
[3] Terasic Technologies Inc. 2013 *DE2i-150 FPGA System User Manual* URL `http://www.terasic.com/`
[4] Putnam A, Caulfield A M, Chung E S, Chiou D, Constantinides K, Demme J, Esmaeilzadeh H, Fowers J, Gopal G P, Gray J, Haselman M, Hauck S, Heil S, Hormati A, Kim J Y, Lanka S, Larus J, Peterson E, Pope S, Smith A, Thong J, Xiao P Y and Burger D 2014 *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on* pp 13–24
[5] Vanderbauwhede W and Benkrid K 2013 *High-Performance Computing Using FPGAs* (Springer)
[6] Cong J, Liu B, Neuendorffer S, Noguera J, Vissers K and Zhang Z 2011 *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* **30** 473–491 ISSN 0278-0070
[7] Menotti R, Cardoso J M P, Fernandes M M and Marques E 2012 *International Journal of Parallel Programming* **40** 262–289
[8] Porto L F and Menotti R 2014 *Anais da 15a edição do Simpósio em Sistemas Computacionais de Alto Desempenho* pp 87–98 ISSN 2358-6613
[9] Martin G and Smith G 2009 *IEEE Design and Test of Computers* **26** 18–25 ISSN 0740-7475
[10] Gupta S, Gupta R, Dutt N and Nicolau A 2004 *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits* (Kluwer Academic Publishers)
[11] Coutinho J G F and Luk W 2003 *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on* (IEEE) pp 278–285
[12] Buyukkurt B, Guo Z and Najjar W 2006 *Reconfigurable Computing: Architectures and Applications* 401–412
[13] Bispo J, Pinto P, Nobre R, Carvalho T, Cardoso J and Diniz P 2013 *Industrial Informatics (INDIN), 2013 11th IEEE International Conference on* pp 602–608
[14] Sangiovanni-Vincentelli A, Yang G, Shukla S, Mathaikutty D and Sztipanovits J 2009 *Design Test of Computers, IEEE* **26** 54–69 ISSN 0740-7475
[15] Cardoso J M, Diniz P C, Petrov Z, Bertels K, Hübner M, van Someren H, Gonçalves F, de Coutinho J G F, Constantinides G A, Olivier B *et al.* 2011 *Reconfigurable Computing* (Springer) pp 261–289
[16] Cardoso J M P, Carvalho T, Coutinho J G F, Nobre R, Nane R, Diniz P C, Petrov Z, Luk W and Bertels K 2013 *Microprocessors and Microsystems - Embedded Hardware Design* **37** 1073–1089 URL `http://dx.doi.org/10.1016/j.micpro.2013.06.001`
[17] Canis A, Choi J, Aldham M, Zhang V, Kammoona A, Anderson J H, Brown S and Czajkowski T 2011 *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays* (ACM) pp 33–36
[18] Lattner C and Adve V 2004 *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* CGO '04 (Washington, DC, USA: IEEE Computer Society) pp 75– ISBN 0-7695-2102-9

[19] Canis A, Choi J, Aldham M, Zhang V, Kammoona A, Czajkowski T, Brown S D and Anderson J H 2013 *ACM Trans. Embed. Comput. Syst.* **13** 24:1–24:27 ISSN 1539-9087 URL `http://doi.acm.org/10.1145/2514740`

[20] Choi J, Brown S and Anderson J 2013 *Field-Programmable Technology (FPT), 2013 International Conference on* (IEEE) pp 270–277

[21] Cardoso J M P 2005 *CF '05: Proceedings of the 2nd conference on Computing Frontiers* (New York, NY, USA: ACM Press) pp 106–115 ISBN 1-59593-019-1

[22] Menotti R, Marques E and Cardoso J M P 2007 *International Conference on Field Programmable Logic and Applications (FPL)* pp 501–502

[23] Menotti R, Cardoso J M P, Fernandes M M and Marques E 2009 *Proceedings of the 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* (Los Alamitos, CA, USA: IEEE Computer Society Press) pp 3–10

[24] Quinlan D *et al.* 2012 Rose compiler infrastructure URL `http://rosecompiler.org/`

[25] Cardoso J M P 2000 *Compilação de Algoritmos em Java para Sistemas Computacionais Reconfiguráveis com Exploração do Paralelismo ao Nível das Operações* Ph.D. thesis Universidade Técnica de Lisboa

[26] Menotti R, Cardoso J M P, Fernandes M M and Marques E 2011 *International Journal of Parallel Programming* 1–28

[27] Malazgirt G, Yantir H, Yurdakul A and Niar S 2014 *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on* pp 1–4

[28] Yantir H, Bayar S and Yurdakul A 2013 *Digital System Design (DSD), 2013 Euromicro Conference on* pp 185–192

[29] LaForest C E and Steffan J G 2010 *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays* FPGA '10 (New York, NY, USA: ACM) pp 41–50 ISBN 978-1-60558-911-4 URL `http://doi.acm.org/10.1145/1723112.1723122`

[30] Guthaus M R, Ringenberg J S, Ernst D, Austin T M, Mudge T and Brown R B 2001 *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop* (Washington, DC, USA: IEEE Computer Society) pp 3–14 ISBN 0-7803-7315-4

[31] Texas Instruments Incorporated 2003 *TMS320C64x Image/Video Processing Library: Programmer's Reference* URL `http://focus.ti.com/lit/ug/spru023b/spru023b.pdf`

[32] Texas Instruments Incorporated 2003 *TMS320C64x DSP Library: Programmer's Reference* URL `http://focus.ti.com/lit/ug/spru565b/spru565b.pdf`