An Empirical Evaluation of High-level Synthesis Languages and Tools for Database Acceleration

Oriol Arcas-Abella*†, Geoffrey Ndu‡, Nehir Sonmez†, Mohsen Ghasempour‡, Adrià Armejach*†,
Javier Navaridas‡, Wei Song‡, John Mawer‡, Adrián Cristal*†§, Mikel Luján‡
*Universitat Politècnica de Catalunya BarcelonaTech (UPC), Barcelona, Spain.
†Barcelona Supercomputing Center (BSC), Barcelona, Spain.
‡School of Computer Science, University of Manchester, Manchester, United Kingdom.
§Centro Superior de Investigaciones Científicas (IIIA-CSIC), Spain.

***Abstract*—High Level Synthesis (HLS) languages and tools are emerging as the most promising technique to make FPGAs more accessible to software developers. Nevertheless, picking the most suitable HLS for a certain class of algorithms depends on requirements such as area and throughput, as well as on programmer experience.**

In this paper, we explore the different trade-offs present when using a representative set of HLS tools in the context of Database Management Systems (DBMS) acceleration. More specifically, we conduct an empirical analysis of four representative frameworks (Bluespec SystemVerilog, Altera OpenCL, LegUp and Chisel) that we utilize to accelerate commonly-used database algorithms such as sorting, the median operator, and hash joins. Through our implementation experience and empirical results for database acceleration, we conclude that the selection of the most suitable HLS depends on a set of orthogonal characteristics, which we highlight for each HLS framework.

I. Introduction

The amount of data in our world is growing rapidly. Processing large volumes of data at high speed is of great interest to companies and organisations. Database queries often make exhaustive use of a significant set of algorithms, some of which are good candidates for FPGA acceleration, in order to substantially improve overall query processing performance [1, 2, 3, 4].

Unfortunately, FPGAs are notoriously difficult to program and even more difficult to debug. Traditional hardware description languages (HDLs) such as VHDL and Verilog lack many of the high-level and abstraction facilities commonly found in modern mainstream languages. As a consequence, the development of hardware can become tedious, inefficient and error-prone for non-expert designers. It might also affect expert FPGA designers from the productivity point of view.

In recent years, several new approaches have been proposed to lower the complexity of hardware development and to make it more attractive to software developers. The most prominent approach is through the use of High-Level Synthesis (HLS) languages and tools, which translate software languages, often C and it variants, into low-level Register-Transfer Level (RTL) descriptions [5, 6, 7, 8]. HLS languages are gaining popularity as they have the potential of "opening FPGAs to the masses". Consequently, FPGA/EDA vendors are increasingly adopting and supporting them. The ease of programmability, performance, resource usage and efficiency can vary from one HLS technology to another, and usually there is a tradeoff between these characteristics.

The main objective of this paper is to undertake a study that analytically and qualitatively compares some of the major, emerging HLS languages in the context of database hardware acceleration using FPGAs. Our study makes the following contributions:

- We select 4 HLS languages and tools for FPGA programming: Bluespec SystemVerilog [9], Altera OpenCL [10], LegUp [11] and Chisel [12]. We justify why we believe that our selection represents most categories of HLS frameworks.
- Based on the literature, we choose 4 algorithms that can accelerate time-consuming database queries, more specifically: bitonic sorting, spatial sorting, the median operator and hash probe. We implement these algorithms using Verilog and the 4 HLS languages, and compare them to analytical models, when applicable.
- We report the results and our experience as programmers using these 4 high-level languages and present the various trade-offs in terms of performance, programmability and resource utilization in the context of developing FPGA accelerators for databases. We show that HLS languages and tools can achieve manually-optimized, RTL-like performance and area results, but not all algorithms are suitable for all HLS languages.

The rest of the paper is organized as follows. The next section provides background for the evaluated languages, where we justify our selection. Section III describes the algorithms to be implemented. We compare our implementations in Section IV, reporting our experience. In Section V we compare the analytical results, and highlight the characteristics of each HLS framework. Section VI summarizes related work and finally, Section VII concludes the paper.

II. Background

HLS tools fill the gap between low-level RTL and high-level algorithms, raising the level of abstraction and effectively hiding the low-level details from the designer. Each proposal stresses a different characteristic (eg., productivity, learning curve, versatility, performance, etc.) resulting in various trade-offs among them.

To classify HLS, we adopted the taxonomy of Bacon *et al.* [6] which defines HLS as any language or tool that includes a high-level feature which RTL does not have. Their classification has five categories: HDL-like languages, CUDA/OpenCL frameworks, C-based frameworks, high-level language-based frameworks and model-based frameworks. In the following subsections, we describe the first four groups and select one language from each for our evaluation. Model-based frameworks are not included in our study because of their specificity to particular domains (eg., DSP modeling).

A. HDL-like HLS: Bluespec SystemVerilog

The first category comprises of modern HDL-like languages, which borrow features from other programming languages to create a new one. This is the case with SystemVerilog [13] and the rule-based Bluespec SystemVerilog (BSV) [9]. We have chose BSV because it is a radically different approach to hardware description, based on guarded rules and syntax inherited from SystemVerilog. In this paradigm, hardware designs are described as data-flow networks of guarded atomic rules. Actions in rules are executed in a transactional manner: state changes happen all-at-once when the rule is fired. Parallelism is achieved through concurrent execution of non-conflicting rules.

B. CUDA/OpenCL HLS: Altera OpenCL

Open Computing Language (OpenCL) is an open industry standard for programming heterogeneous computing platforms (a host CPU, GPU, DSP or FPGA). It is based on standard ANSI C (C99) with extensions to create task-level and data-level parallelism. Altera's SDK for OpenCL (AOCL) [10] exploits parallelism in data-independent threads, or "work items" in OpenCL speak. AOCL translates the software description into a pipelined hardware circuit, where each stage of the pipeline executes a different thread. This approach is less versatile than general-purpose C compilers, but can be more efficient for data-flow and streaming applications, which is one of the drawbacks of other HLS. We have included this tool in our study because we believe it is representative of HLS tools targeted at streaming problems, such as Impulse C or those based on CUDA.

C. C-based HLS: LegUp

The other categories in Bacon *et al.*'s classification are frameworks that target subsets, or extensions of already-existing software languages. In most of the cases, the designers adopt a popular language to smoothen the learning curve. The most prominent group is based on C: LegUp [11], ROCCC [14] and Impulse C (specialized in stream programming) [15] support C subsets. xPilot [16] (now Xilinx Vivado [17]) and Calypto Catapult C [18] also accept C++ and SystemC.

We included LegUp in our evaluation for two reasons. First, (i) it is open-source, and (ii) we believe that the synthesis mechanism is similar to those used in other C-based HLS tools. Many C-based HLS tools perform hardware synthesis after transforming C into an intermediate representation, usually using external tools such as LLVM [19]. LegUp compiles LLVM code into Verilog. The C functions are converted into Finite State Machines (FSM). Local and global variables are stored in shared memories (Block RAMs or external DDR), and are accessed by the FSMs, which load, modify and store the data following the algorithm. The efficiency of this approach is based on executing independent LLVM instructions concurrently, and other advanced techniques such as loop unrolling and pipelining (with certain limitations).

D. High-level Language Frameworks: Chisel

This last group includes frameworks that translate high-level languages (other than C) into hardware. Some examples are the event-driven Esterel [20], Kiwi [21] (C#) and Lime [22] (Java). Chisel [12] is based on the functional language Scala (which is based on Java), and therefore targeted to high-productivity. The hardware designs are pure Scala applications and can be synthesized into C++ simulators or Verilog RTL descriptions. The framework is made with Scala, and provides basic data types, structures and language constructs. However, the interconnection of the elements (ie., modules, wires, registers) is done in an RTL-like manner.

III. Studied Database Algorithms

Our comparisons focus on three common and time-consuming database operations: sorting, aggregation and joins. The inherent parallelism of sorting makes it suitable for efficient FPGA implementations [23]. We chose two sorting algorithms that are suitable for hardware implementation, namely bitonic and spatial sorters. For aggregation, we implemented the median operator with a sliding-window, also used in [3]. Finally, for table joins, we included a hash probe algorithm to accelerate hash join operations [4].

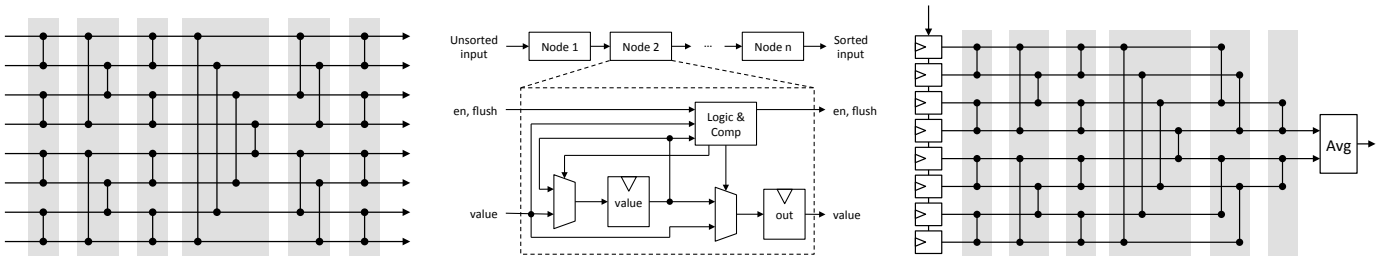
A. Bitonic Sorter

A bitonic sorter is a type of a sorting network [24] particularly efficient in hardware, consisting of multiple levels of compare-and-exchange units. The sorting is performed in $\mathrm{O}(\log^2 n)$ time complexity, and requires $\mathrm{O}(n \log^2 n)$ comparators that can be pipelined, increasing the frequency and the throughput. Figure 1a shows an 8-input bitonic sorter. Such a sorting network is straightforward to write in almost any language. It can be expressed recursively and composed together to form larger networks. Since it produces sorted sets every cycle, it is appropriate for high-bandwidth I/O interfaces, in particular parallel ones. However, this parallelism also results in high resource usage.

B. Spatial Sorter

The spatial sorter [25] is composed of an array of sorting registers, each of which effectively does a compare and swap operation [1]. As seen in Figure 1b, the main ingredients of a sorter node are a comparator, two registers and two multiplexers. New elements are inserted at the beginning of the sorter array. At each clock cycle and on each node, an input value is compared with the current value. The greater value is stored in the sorter node, and the smaller value is passed to the next node.

The spatial sorter has a worst case time cost of $2n$ cycles to sort an input set of size n. After $2n$ cycles, the sorted set starts to be emitted by the last node. In order to accept an input and



(a) 8-input Bitonic sorter. (b) Spatial sorter implementation. (c) Median operator (8-element sliding window).

Fig. 1: The Bitonic sort, spatial sort and median operator algorithms. In sorting networks the horizontal arrows are input values, and the vertical lines are the Knuth compare-and-exchange operator ( ). Sorting stages are shaded in gray.

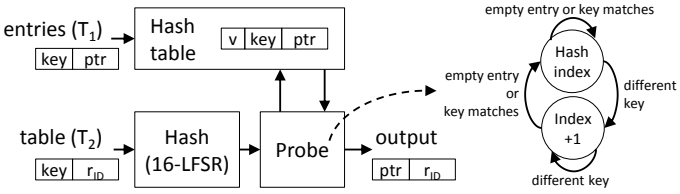


Fig. 2: Hash probe algorithm. Detail of the state machine.

to produce a sorted output at each clock cycle, this mapping-friendly sorting infrastructure could simply be duplicated, to take turns processing the inputs, and not to stall. It can also be coupled with merging circuitry to merge-sort even larger sets, as shown in [25]. The resource usage of the spatial sorter is proportional to its number of inputs, whereas the bitonic sorter grows supra-linearly. Therefore, the spatial sorter is more appropriate for lower bandwidth I/O interfaces.

C. Median Operator

Aggregation operations, which reduce a stream of data to a single result, are ubiquitous in database queries. We implemented the median operator as in Mueller *et al.* [3], expressed by the CQL [26] query Q_1:

```
SELECT median(v) FROM S [ Rows 8 ].   (Q1)
```

This expression describes a median operator over a sliding window of 8 elements. Figure 1c depicts our implementation using a bitonic sorter. For every cycle and for each new element inserted into the window, a new median value is calculated. Since only the median values of the sorter are used, some comparators and registers are optimized away by the synthesis tool. We used a 16-input window for evaluating this algorithm.

D. Hash Probe

Database operations that join two tables using a common column are frequent and time-consuming. The hash join algorithm consists of first building a hash table using the smaller table (T_1) and then probing it with all the rows of the larger table (T_2). In our implementation, each hash table entry contains the matching key and a pointer (`ptr`) to a list of all the rows with that key in T_1. The hash function was implemented as a Linear Feedback Shift Register (LFSR). If the hash indexes of two keys collide, the latter key is inserted in the next consecutive empty bucket of the hash table.

We preload the hash table into the BRAMs of the FPGA. Then, for each (`key`, $\texttt{row}_{\texttt{ID}}$) input tuple of table T_2, a hash index is calculated and the hash table is probed for matches. Starting from that index, all the non-empty consecutive entries of the hash table are checked. If the keys in the hash table entry and the T_2 tuple match, the result (T_1 `ptr`, T_2 $\texttt{row}_{\texttt{ID}}$) is output. If an empty bucket is found, the current T_2 tuple does not exist in the hash table and is skipped.

In contrast to previous algorithms that require more computational power, hash probe hardware essentially consists of a memory that is randomly accessed and some control logic. In Figure 2 we show the main elements of the hardware model. The `Probe` state machine implements the hash table probe algorithm. The 16-iteration LFSR-based hash function can be pipelined to allow constant throughput.

IV. Programming Experience Evaluation

In this section, we share our experience implementing the algorithms using the different languages. For sorting algorithms, the input data is 32-bit key – 32-bit value pairs (ie., 16 × 64 = 1,024 bits in size). The median operator uses 32-bit key inputs. The hash probe uses 16-bit key – 32-bit value pairs, and a hash table with 64K × 64-bit buckets (512 KB) and a load factor of 0.6. The size of (T_1) is 400 MB and the size of (T_2) is 600 MB. None of the tools required more than 30 seconds to generate the Verilog descriptions.

A. Bluespec SystemVerilog

All the algorithms were substantially easy to describe using the data-flow, rule-based paradigm of BSV. The advanced evaluation system of BSV handles well the recursive definition of the bitonic sorter. The BSV models are fully parametrized, and can generate hardware models for arbitrary input sizes, as well as to perform a different number of comparisons at each pipeline stage. In Figure 3a, we show one of the submodules of the bitonic sorter in BSV.

Implementing hash probe in BSV proved to be more difficult than in Verilog. Obtaining an optimal scheduling in BSV can sometimes require some extra effort from the designer, due to the strict sequentially consistent paradigm of the language, which might not be obvious to designers with

```
module mkSorter (SortBox#(n,t));
  ...
  rule do_bitonic_sort;
    Integer stage = 0;
    for (Integer ai=2; ai <= valueOf(n); ai=ai*2)
      for (Integer bi=ai; bi > 1; bi=bi/2) begin
        let x = regs[stage];
        let y = (stage+1 < valueOf(num_stages)) ?
          regs[stage+1] : destW;
        for (Integer i=0; i < valueOf(n); i=i+bi)
          for (Integer j=0; j < bi/2; j=j+1) begin
            let k2 = bi!=ai?(i+j+bi/2):(i+bi-j-1);
            Bool swap = compareData(x[i+j], x[k2]);
            y[i+j] <= (swap ? x[k2] : x[i+j]);
            y[k2] <= (swap ? x[i+j] : x[k2]);
          end
        stage = stage + 1;
      end
  endrule
  ...
endmodule
```

(a) Example Bluespec SystemVerilog code

```
__kernel
__attribute__((
  reqd_work_group_size(WORKGROUP_SIZE,1,1)))
void bitonic(
    __global int2* restrict input_data,
    __global int2* restrict output_data) {
  const unsigned group = get_group_id(0)*16;
  int2 temp[16];
  #pragma unroll
  for (unsigned i = 0; i < 16; ++i)
    temp[i] = input_data[group+i];
  compAndSwap(&temp[0], &temp[1]);
  compAndSwap(&temp[2], &temp[3]);
  compAndSwap(&temp[4], &temp[5]);
  compAndSwap(&temp[6], &temp[7]);
  ...
  #pragma unroll
  for (unsigned i = 0; i < 16; ++i)
    output_data[group+i] = temp[i];
}
```

(b) Example Altera OpenCL code

```
volatile int input_data[16];
volatile int output_data[16];
int main() {
  int i, temp[16];
  for (i = 0; i < 16; ++i)
    temp[i] = input_data[i];
  compAndSwap(&temp[0], &temp[1]);
  compAndSwap(&temp[2], &temp[3]);
  compAndSwap(&temp[4], &temp[5]);
  compAndSwap(&temp[6], &temp[7]);
  ...
  for (i = 0; i < 16; ++i)
    output_data[i] = temp[i];
  return 0; }
```

(c) Example LegUp code

```
class Bitonic(val n:Int,k:Int,d:Int)
    extends Module {
  val io = new BitonicIfc(n, k, d)
  ...
  val subu = Vec.fill(2) {
    Module(new Bitonic(n/2, k, d)).io }
  for (t <- 0 until n/2 ) {
    subu(0).in(t) := io.in(t)
    inputs0(t) := subu(0).out(t)
    subu(1).in(t) := io.in(t+n/2)
    inputs0(t+n/2) := subu(1).out(t)
  }
  ...
}
```

(d) Example Chisel code

Fig. 3: Bitonic sorter code snippets for each HLS framework

an RTL background. Although Bluespec adopts many syntax expressions from SystemVerilog, the learning curve is steep even for developers with background in hardware design. The type system is powerful but complex, and the programmer must think in terms of rules and dataflow. However, once a critical knowledge level is reached, the productivity can be very high. Another benefit is the strict control over the timing model, which is not possible in C-based HLS frameworks.

B. Altera OpenCL

Altera OpenCL is strongly influenced by GPU programming. However, our implementation of the bitonic sorter is slightly different from the typical GPU implementation to enable us to fully exploit parallelism on the FPGA. Figure 3b shows a snippet of the OpenCL code. The code to be accelerated ("kernel" in OpenCL speak) contains the hard-coded compare-and-exchange operations of the 16-input bitonic sorter. We had to hard-code the 16-input version of the bitonic sorter, as we did for LegUp. The software region to be accelerated is marked with the `__kernel` keyword, which is pipelined by AOCL. The programmer can use directives, such as the `#pragma unroll`, to guide the compiler.

The spatial sorter was more difficult to implement than the bitonic sorter, as in LegUp, even with the multithreading capabilities of OpenCL. In our implementation, each work-item fetches an incoming value from the global memory and informs the master thread (work-item 0) about its position in the array. The master thread is responsible for appropriately moving data between the threads. At the end of the loop iteration, the sorted data is written back to global memory. The calculation of the median operator is performed by work-groups of threads. As work-groups are mutually independent, each one needs to have its own input buffers to avoid conflicts. Similarly to LegUp, the hash probe algorithm was ported easily to OpenCL. But in this case, we used a sequential implementation and relied on AOCL to pipeline the design and to optimize memory accesses.

We found that AOCL has a steeper learning curve than LegUp. The programmer needs considerable knowledge about underlying OpenCL concepts such as work-groups and work-items. On the other hand, the efficiency obtained can be much higher for some classes of database problems, as we describe in the next section.

TABLE I: Analytical models for the resource usage of the studied algorithms.

Algorithm	Stages (S) Value	Stages (S) Complexity	Registers Value	Registers Complexity	Logic Value	Logic Complexity
bitonic	$\sum_{i=1}^{\log n} i$	$\mathrm{O}\left(log^2 n\right)$	nSw	$\mathrm{O}\left(nw \log^2 n\right)$	$\frac{nSw}{2}$	$\mathrm{O}\left(nw \log^2 n\right)$
spatial	n	$\mathrm{O}\left(n\right)$	$2nw$	$\mathrm{O}\left(nw\right)$	$2nw$	$\mathrm{O}\left(nw\right)$
median	$\sum_{i=1}^{\log n} i$	$\mathrm{O}\left(\log^2 n\right)$	$nSk - 2k\sum_{i=0}^{\log n-2}\left(\frac{n}{2} - 2^i\right)$	$\mathrm{O}\left(nk \log^2 n\right)$	$\frac{nSk}{2} - k\sum_{i=0}^{\log n-3}\left(\frac{n}{4} - 2^i\right)$	$\mathrm{O}\left(nk \log^2 n\right)$

Note: k = key bits, v = value bits, $w = k + v$. Median only uses k.

C. LegUp

The main advantage of pure-C HLS tools is that the learning curve is very smooth, as most programmers are already familiar with C. On average, the algorithms required very few lines of code. Moreover, most already-existing algorithms written in C can be ported to an FPGA almost seamlessly. However, obtaining an efficient implementation requires experience using the tools, as well as prior knowledge on the target technology to later optimize it. In addition, a substantial rewriting of the initial code may be required.

In Figure 3c, we show the most interesting fragments of our C version of the bitonic sorter. The LegUp code, which can be compiled as regular C code and executed on any processor, is completely straightforward to a C programmer. The resulting binary is functionally equivalent to the hardware generated by LegUp. This feature is interesting for fast simulation and debugging, as well as for the migration of software kernels to an FPGA. On the other hand, the C language has a limited evaluation system, the C preprocessor, based on conditional directives and macros. In this sense, Verilog allows recursive and iterative code generation, and Chisel and BSV have advanced evaluation systems. Due to the limitations of the C preprocessor, we hard-coded the 16-input bitonic algorithm, which makes the code not parametrizable and less reusable. Another option would be having the compiler unroll the parameterized structures, such as loops. Unfortunately, the current version of LegUp has limited support for nested loops or array-based dependencies (where the destination and the source are in the same array).

We had to learn how to correctly describe the hardware using C. For instance, input and output buffers (`input_data` and `output_data`) are marked as `volatile` to indicate the compiler not to try optimizing away memory accesses. Instead of operating over data in the `main` function the data is copied into a temporal buffer. This will make the compiler read all the input data and optimize the sorting over the temporal buffer. The spatial sorter, very natural to express in any HDL, is not well suited for C-based HLS. We found that the multi-threaded nature of the algorithm, where independent sorting units exchange data, is very difficult to express in C. The implementation of the median operator was more straightforward. Storing only the median value (the average of the two middle values of the sorted set) allows the LegUp compiler to optimize away the extra computation and the LLVM compiler to trim the unused data paths.

We saw that hash probe is a very suitable algorithm to be expressed in C-based HLS, yielding the best overall performance results for LegUp, as shown in the next section.

D. Chisel

The programming model of Chisel is very similar to RTL languages. Hardware units are defined and interconnected in an imperative way. However, for evaluating the code, all the high-level constructs of the Scala language are available. In Figure 3d we show an example code snippet of our bitonic implementation in Chisel.

As in the case of BSV, almost all the examples were very easy to express as parameterizable implementations, and the high-level constructs produced very succinct code. One special case was the Chisel implementation of hash probe, which was easier than BSV (as we did not run into control flow issues) and Verilog (as we were able to use higher level constructs). Being a subset of Scala, Chisel is a good language for developers with some background on Java. It supports advanced features like polymorphism and parametrized modules. However, we believe that some features would improve productivity even further. For instance, BSV-like implicit condition handling would simplify the control logic of designs.

V. Empirical Evaluation and Comparison

In this section, we compare the empirical results against analytical models and hand-written Verilog models. We targeted the implemented algorithms to an Altera Stratix V 5SGXA7 FPGA, with the same synthesis options. We used the Quartus "Early Timing and Area Estimates" flow to compile the designs.

The performance of an HLS can be seen as the combination of the algorithmic performance of the hardware designed and the I/O performance. In this paper, we concentrate on algorithmic performance, however the I/O performance also has to be taken into account when choosing an HLS. The biggest advantage of using Altera OpenCL is its ability to automatically generate I/O interfaces with the host. Bluespec provides libraries for interfaces such as Ethernet or PCIe. LegUp allows interfacing a soft CPU core (hybrid flow) and Chisel does not have support for I/O interfacing yet.

A. Analytical Analysis

To have a concrete baseline for comparison, we first performed an analytical analysis of the expected FPGA resource utilization for determining the number of registers and LUTs needed to implement our algorithms.

In an efficient implementation, the manually-optimized Verilog model should directly match the analytical model. The resource usage of the BSV and Chisel designs is also expected

TABLE II: Evaluation of the four algorithms on different programming environments.

Implementation	Registers FF	% incr.	Logic LUT	% incr.	ALM*	Memory (Kbits)	F_{max} (MHz)	Throughput MB/s	%	LoC
bitonic analytical	10,240	0.00%	2,560	0.00%						
bitonic Verilog	10,250	0.10%	2,640	3.13%	7,210.8	0.00	311.43	38,016.36	100.00%	134
bitonic BSV	10,250	0.10%	2,640	3.13%	6,997.5	0.00	313.97	38,326.42	100.82%	57
bitonic Chisel	10,272	0.31%	2,649	3.48%	5,571.0	0.00	314.96	38,447.27	101.13%	114
bitonic LegUp	4,210		5,180		3,973.4	0.00	211.86	1,034.47	2.72%	101
bitonic OpenCL	38,455		5,221		15,842.6	361.38	307.12	1,317.21	3.46%	140
STL sort C++ (host CPU)							2,300.00	570.42	1.50%	3
spatial analytical	2,048	0.00%	640	0.00%						
spatial Verilog	2,081	1.61%	641	0.16%	1,359.5	0.00	341.30	1,301.96	100.00%	98
spatial BSV	2,112	3.13%	1701	165.75%	1,081.0	0.00	343.52	1,310.42	100.65%	181
spatial Chisel	2,112	3.13%	720	12.50%	1,053.0	0.00	345.30	1,317.21	101.17%	87
spatial LegUp	1,115		823		612.5	0.50	309.12	3.13	0.24%	28
spatial OpenCL	26,059		14,667		15,072.3	877.84	236.85	660.53	50.73%	66
median analytical	4,544	0.00%	2,240	0.00%						
median Verilog	4,555	0.24%	2,352	5.00%	4,009.5	0.00	302.76	1,154.94	100.00%	159
median BSV	4,554	0.22%	6,168	175.36%	3,359.5	0.00	334.67	1,276.66	110.54%	70
median Chisel	4,577	0.73%	2,351	4.96%	3,321.5	0.00	338.98	1,293.11	111.96%	132
median LegUp	10,449		5,262		3,781.4	0.47	174.98	34.25	2.97%	97
median OpenCL	19,366		7,590		9,309.5	190.06	312.10	920.60	79.71%	84
median C++ (host CPU)							2,300.00	836.10	72.39%	6
hash probe Verilog	995		174		327.5	3,136.00	174.06	995.98	100.00%	66
hash probe BSV	1,150		166		365.5	3,136.00	181.46	1,038.32	104.25%	124
hash probe Chisel	1,020		179		333.5	4,096.00	171.59	981.85	98.58%	83
hash probe LegUp	345		397		262.5	4,096.00	302.85	61.90	6.22%	50
hash probe OpenCL	35,536		21,854		19,175.6	3,876.08	270.19	2.14	0.21%	59
hash probe C++ (host CPU)							2,300.00	433.32	43.51%	18

*ALM (Adaptive Logic Module): Altera's basic cell blocks, with an 8-input fracturable LUT and four 1-bit registers.

to be very close to the Verilog and analytical baselines. The LegUp and the AOCL models follow a different computational paradigm that would be very difficult to model, so we will not compare them against the analytical models. Similarly, no analytical model was devised for the resource usage of the hash probe design, which is mostly made up of control logic.

In Table I, we show the number and complexity of the stages, registers and combinational logic of each algorithm. The parameter n represents the size of the sorting set. The parameters k and v represent the key and value sizes in bits (and $w = k + v$). It can be seen that the Bitonic sorter needs $\mathrm{O}\big(nw \log^2 n\big)$ registers and $\mathrm{O}\big(nw \log^2 n\big)$ combinational LUTs (for the comparators). In the case of the median operator, the costs are $\mathrm{O}\big(nk \log^2 n\big)$ because only the keys are sorted, and only the middle numbers of the sorting are used, allowing to optimize away some registers and comparators, as shown in Figure 1c. The spatial sorter requires 2 registers in each sorting unit: one for the current value and one to store the outgoing one. The hardware model requires $\mathrm{O}\big(nw\big)$ registers and $\mathrm{O}\big(nw\big)$ combinational LUTs.

B. Experimental Results

In Table II we show the empirical results for all the languages and tools evaluated. For each algorithm, we show the resource usage, maximum frequency, estimated throughput (MB/s) and lines of code needed (LoC)[1]. The LegUp resource usage was obtained by stripping out the additional infrastructure generated by the tools, and only leaving out the algorithmic kernel. In the AOCL implementations complex I/O optimizations are implemented, like input and output buffering, resulting in an intensive resource usage. Additionally, RAM blocks were extensively used by the OpenCL implementations, and minimally used by LegUp, as well. The block memory usage for hash probe shows that for some implementations it was possible to optimize away the unused bits (only 49 bits of the 64-bit hash table buckets were used).

In terms of performance, our results demonstrate that HLS tools are indeed able to offer competitive performance to fine-tuned Verilog/VHDL, effectively accelerating database operations. We also implemented these algorithms in software, using the efficient Standard Template Library (STL) implementations of C++, running on a host machine with Intel Xeon E5-2630 CPU at 2.3 GHz. We made sure that the benchmarks used already-cached data, attempting to mimic ideal conditions in software. The results show that the computational power of most of the HLS implementations is significantly higher than a software version. Furthermore, we used 16-input designs in this work, while FPGAs allow bigger circuits to be implemented, and higher performance gains can be expected (along with considerable power savings compared to a high-performance CPU).

The throughput results can be thought of as being interfaced through BRAMs. For AOCL, we derived the unconstrained throughput using the de-rate factor that the compiler applies when the maximum bandwidth is exceeded. The high bandwidth achieved by bitonic could be provided by DRAM, as available in the Maxeler MAX3 platform (38.4GB/sec [27]). For other I/O interfaces that provide less throughput, some

[1] Although we consider that LoC cannot be used as the primary criteria, and more advanced metrics should be used such as function points. The creation of a *hardware description sizing metric* adapted to HDLs and HLS is one of the possible future research directions.

options are: (i) to generate a smaller/slower circuit that requires less bandwidth and saves unused computational power, (ii) to employ caching structures similar to ROCCC's smart buffers [14], or (iii) to use another algorithm, such as the spatial sorter instead of bitonic.

The bitonic sorter is easy to express in all languages, and delivers a speedup between 1.81x and 67.4x over the software version. The C-based HLS tools exhibit diverse behaviors. AOCL outperforms the software in most of the cases and achieves competitive results in the spatial sorter and the median operator ($>$ 50% of the hand-coded Verilog throughput). In addition, AOCL can automatically replicate the computation units, resulting in linear speedup in our experiments, at the cost of more resource and bandwidth (in Table II we only used 1 computational unit). LegUp has moderate throughput results, but requires very few resources.

For the database algorithms that we have studied, BSV and Chisel produced code that is on par with hand-optimized Verilog, yielding the best throughput-per-area ratios. Curiously, the compiler might even be able to use some extra logic and to optimize further in certain cases, as in the median operator implementations in BSV and Chisel. In the case of BSV, the high LUT usage for the spatial sorter and the median operator (caused by the rule-based programming model) is absorbed by the ALMs and doesn't result in a higher area requirement. The register usage increase for Verilog, BSV and Chisel over the analytical models were mostly caused by control logic.

C. Discussion

With our experience and experiments, we can conclude that there is no obvious election when choosing an HLS, but an orthogonal set of characteristics that must be considered. Bluespec SystemVerilog has a steep learning curve, but provided good performance results in our experiments. Among other benefits, it guarantees tight control over the cycle accurate model and automatic flow control validation, and supports high-level, parameterizable constructs. It is a good choice to implement system-level HW models, especially for designers with a background in RTL design or in Haskell.

The Altera's OpenCL framework required succinct implementations while delivering good throughput rates on some algorithms, and was easy to use because the OpenCL standard is based on C. However, we found it difficult to parameterize the designs and the resource footprint was the highest (as a result of including a complex I/O infrastructure automatically). It is a natural choice for data-flow algorithms, especially when accelerating already-existing C kernels, but some expertise in OpenCL is required.

The LegUp HLS tool accepts generic C code. We consider it the easiest to learn and to use. Its performance results were lower than the other tools considered (which may improve in future versions). Thus, it can be considered for designers from any field with little experience using HDLs, and it allows to easily implement algorithms with lower bandwidth requirements, predominance of flow control structures, or to accelerate already-existing C code.

Finally, Chisel delivered good performance results from relatively succinct implementations. As in BSV, it retains cycle accuracy and the high-level Scala constructs allow to parameterize the code. However, it lacks a proper standard type library, and the hardware scheduling/interconnection must be done manually as in RTL-like languages. It is a good choice for designers with a strong RTL background (and some Scala knowledge), enabling them to implement system-level designs.

VI. Related Work

An exhaustive survey of HLS tools has been published recently by Daoud *et al.* [7], describing a plethora of HLS tools from the last 30 years, but it does not provide any empirical evaluation or comparison between them. Bacon *et al.* 's classification of HLS frameworks [6] is also very rich in technical details, but no direct comparison is made between different implementations. Cong *et al.* [8] provide a comparison of development time and resource usage in Autopilot against hand-written RTL with two examples, on an optical flow algorithm and a wireless application.

Many studies focus on a single language in order to evaluate its benefits compared to RTL. Bachrach *et al.* [12] present Chisel and compare it against Verilog to demonstrate that better performance with less lines of code can be obtained when implementing a RISC CPU. Cornu *et al.* [5] compare a genetic sequence algorithm on Impulse C and RTL implementations. They show a 4.2$\times$ speedup over hand-written RTL code, and state that HLS may provide higher performance than RTL because higher amounts of optimization can be obtained from high-level design, while low-level optimization is less efficient for the designer.Agarwal *et al.* [28] compare BSV against a C-based HLS tool, implementing a complex Reed-Solomon decoder. They show that the latter could have limited performance and high resource usage, while BSV can obtain better performance, requiring similar resources as a commercial IP. Meeus *et al.* [29] provide an overview of many HLS tools (mostly C, Matlab and BSV), and a qualitative comparison of the Sobel edge detector image processing algorithm. They mainly focus on tool comparison, on metrics like area, learning curve, and the ease of implementation, but not on performance.

Hammami *et al.* [30] present a comparison of 4 benchmark designs (two filters, FFT and ray casting), which are implemented on 3 C-based HLS (ImpulseC, Handel-C and SystemC). The paper which does a detailed comparison of area, throughput and tool variation, also looks into how the tools deal with concurrency and pipeline extraction. Virginia *et al.* [31] include three different C-to-VHDL compilers (ROCCC, SPARK and their DWARV proposal) and compare a large subset of kernels using metrics such as throughput-per-slice, as well as readability, writing effort, supported ANSI-C subset, testability and hardware knowledge required. They conclude that the restrictions on the supported C subset directly influence the rewriting effort that is needed to make certain kernels compatible with the compilers.

Hara *et al.* [32] propose a complete benchmark suite for HLS, with complex and diverse applications. However, it is specific to C-based languages. Database acceleration using FPGAs has been studied extensively before. Mueller *et al.* [3] presented sorting networks and the same median operator with sliding window that we use in this work. Halstead *et al.* [4] accelerate join operations using an FPGA, where columns with

identical values are chained together in an address table and the hash probe is done in parallel. Istvan et al. present an FPGA-based hash table design with chaining and parallel lookups for a memcached server [33]. Dennl *et al.* [2] accelerate SQL projections (`SELECT`) and restrictions (`WHERE`) using dynamic reconfiguration. Finally, Koch *et al.* [1] implement a space-efficient merge sort and make use of partial reconfiguration to outperform GPUs and the Cell processor.

VII. Conclusions

Database management systems present many opportunities for hardware acceleration, especially under very large workloads. In this paper, we have presented an empirical evaluation of four HLS frameworks (Bluespec SystemVerilog, Altera OpenCL, LegUp and Chisel), while implementing four algorithms relevant to database acceleration. We have justified the representativity of these languages and tools among the HLS diversity. We also described our experience as programmers using these different HLS languages and tools.

Choosing the most appropriate HLS framework depends on a set of requirements including performance and resource footprint, as well as a number of contextual conditions such as the programmer's background. We found out that there is no single obvious election. Therefore, we provide a detailed discussion that highlights which of these orthogonal goals each HLS tool is the most appropriate for. The shortage of similar studies makes further research necessary. We encourage other researchers to download the source code of our experiments from `https://github.com/bsc-uniman/fpl14-hls-eval` to investigate more efficient implementations, or to extend them with new languages and tools. We believe that "hardware description sizing" metrics and HLS benchmark suites (beyond C-based languages) are first-priority research directions.

Acknowledgements

We want to thank Osman S. Unsal and Behzad Salami for their contributions, and the FPL reviewers for their feedback. The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 318633, and UPC project TIN2012-34557. Luján is supported by a Royal Society University Research Fellowship and Mawer is funded by the EPSRC programme grant PAMELA EP/K008730/1.

References

[1] D. Koch and J. Torresen, "FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting," in *Proc. 19th isFPGA*. ACM, 2011, pp. 45–54.

[2] C. Dennl, D. Ziener, and J. Teich, "On-the-fly composition of FPGA-based SQL query accelerators using a partially reconfigurable module library," in *Proc. FCCM, IEEE*, 2012, pp. 45–52.

[3] R. Mueller, J. Teubner, and G. Alonso, "Data processing on FPGAs," *VLDB Endowment*, vol. 2, no. 1, pp. 910–921, 2009.

[4] R. J. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer, "Accelerating join operation for relational databases with FPGAs," in *Proc. FCCM*, 2013, pp. 17–20.

[5] A. Cornu, S. Derrien, and D. Lavenier, "HLS tools for FPGA: Faster development with better performance," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2011, pp. 67–78.

[6] D. F. Bacon, R. Rabbah, and S. Shukla, "FPGA programming for the masses," *Communications of the ACM*, vol. 56, no. 4, pp. 56–63, 2013.

[7] L. Daoud, D. Zydek, and H. Selvaraj, "A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing," in *Advances in Systems Science*, 2014, pp. 483–492.

[8] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. on Computer-Aided Design of ICs and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[9] "Bluespec, Inc." http://www.bluespec.com.

[10] *Altera SDK for OpenCL Programming Guide Version 13.0 SP1*, 2013.

[11] A. Canis *et al.*, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proc. 19th isFPGA*, 2011, pp. 33–36.

[12] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proc. 49th DAC*, 2012, pp. 1216–1225.

[13] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," *IEEE STD 1800*, pp. 1–1285, 2009.

[14] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers, "Optimized generation of data-path from c codes for FPGAs," in *Proceedings of Design, Automation and Test in Europe-Volume 1*, 2005, pp. 112–117.

[15] "Impulse Accelerated Technologies, Inc." http://www.impulseaccelerated.com/.

[16] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "xpilot: A platform-based behavioral synthesis system," *SRC TechCon*, vol. 5, 2005.

[17] "Xilinx Vivado ESL Design," http://www.xilinx.com/pro-ducts/design-tools/vivado/integration/esl-design/.

[18] "Calypto Catapult C," http://calypto.com/en/products/catapult/overview.

[19] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Int. Symp. on Code Generation and Optimization*, 2004, pp. 75–86.

[20] G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation," *Science of computer programming*, vol. 19, no. 2, pp. 87–152, 1992.

[21] D. Greaves and S. Singh, "Kiwi: Synthesis of FPGA circuits from parallel programs," in *16th Int. Symp. on FCCM*, 2008, pp. 3–12.

[22] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a java-compatible and synthesizable language for heterogeneous architectures," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 89–108, 2010.

[23] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: high performance graphics co-processor sorting for large database management," in *Proc. SIGMOD*. ACM, 2006, pp. 325–336.

[24] K. E. Batcher, "Sorting networks and their applications," in *Proc. spring joint computer conference*. ACM, 1968, pp. 307–314.

[25] A. Parashar *et al.*, "Triggered instructions: A control paradigm for spatially-programmed architectures," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 142–153, Jun. 2013.

[26] A. Arasu, S. Babu, and J. Widom, "The CQL Continuous Query Language: Semantic Foundations and Query Execution," Stanford InfoLab, Technical Report, 2003.

[27] "Maxeler MaxCloud," http://www.maxeler.com/products/maxcloud/.

[28] A. Agarwal, M. C. Ng *et al.*, "A comparative evaluation of high-level hardware synthesis using reed–solomon decoder," *Embedded Sys. Letters, IEEE*, vol. 2, no. 3, pp. 72–76, 2010.

[29] W. Meeus, K. VanBeeck, T. Goedem, J. Meel, and D. Stroobandt, "An overview of todays high-level synthesis tools," *Design Automation for Embedded Systems*, vol. 16, no. 3, pp. 31–51, 2012.

[30] O. Hammami, Z. Wang, V. Fresse, and D. Houzet, "A case study: Quantitative evaluation of c-based high-level synthesis systems," *EURASIP J. Emb. Sys.*, 2008.

[31] A. J. Virginia, Y. D. Yankova, and K. L. Bertels, "An empirical comparison of ANSI-C to VHDL compilers: SPARK, ROCCC and DWARV," in *Annual Workshop on Circuits, Systems and Signal Processing ProRISC*, 2007, pp. 388–394.

[32] 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," vol. 17, 2009, pp. 242–254.

[33] Z. István, G. Alonso, M. Blott, and K. A. Vissers, "A flexible hash table design for 10gbps key-value stores on fpgas," in *FPL*, 2013, pp. 1–8.