Automating Data Reuse in High-Level Synthesis

Wim Meeus
Imec and Ghent University, Gent, Belgium
Email: Wim.Meeus@UGent.be

Dirk Stroobandt
Ghent University, Gent, Belgium
Email: Dirk.Stroobandt@UGent.be

***Abstract*—Current High-Level Synthesis (HLS) tools perform excellently for the synthesis of computation kernels, but they often don't optimize memory bandwidth. As memory access is a bottleneck in many algorithms, the performance of the generated circuit will benefit substantially from memory access optimization.**

In this paper we present an automated method and a toolchain to detect reuse of array data in loop nests and to build hardware that exploits this data reuse. This saves memory bandwidth and improves circuit performance. We make use of the polyhedral representation of the source program, which makes our method computationally easy. Our software complements the existing HLS flows. Starting from a loop nest written in C, our tool generates a reuse buffer and a loop controller, and preprocesses the loop body for synthesis with an existing HLS tool. Our automated tool produces designs from unoptimized source code that are as efficient as those generated by a commercial HLS tool from manually-optimized source code.

I. Introduction

High-level Synthesis (HLS) is a recent step in the design flow of a digital electronic circuit, moving the design effort to higher abstraction levels [1], [2]. HLS translates a behavioral description of an algorithm at the algorithmic abstraction level into a design at the Register Transfer Level (RTL). Ref. [3] shows that some of these tools generate excellent RTL designs for computation kernels, sometimes outperforming handcrafted RTL design in terms of speed, chip area and power consumption. Most existing HLS tools don't optimize memory accesses though, leaving a lot of potential for circuit performance optimization untapped. This is particularly unfortunate because the communication between processor cores and memory is a well-known bottleneck that limits performance.

We present an automatic method for generating data reuse buffers during HLS. This should lead to a more optimal design than a general-purpose memory system with caches and scratchpad memories. We focus on array accesses in loop nests, because that is where the most gain can be obtained. We ensure that every array location is read and/or written only once during the execution of the loop nest. This is known as communication coalescing and leads to a reduction of the data traffic to and from memory, improving circuit performance and alleviating the memory bottleneck. Our tool produces RTL code for reuse buffers and the loop controller, and preprocessed C code of the loop body for the generation of a datapath using an existing HLS tool. Experiments reveal that the performance of designs generated using our automated flow with unoptimized source code match that of traditional HLS using hand-optimized code. The main contributions of this work are:

- An automated method which harnesses the polyhedral representation of the source program to discover data reuse in an algorithm.
- A method for efficiently exploiting data reuse using data reuse buffers, alleviating the memory bottleneck. The use of the polyhedral representation leads to an elegant design with linear memory access patterns.
- An automated design flow to implement said reuse buffers in hardware in the form of an RTL design, together with a controller that synchronizes memory accesses, the reuse buffer and the datapath.

II. The polyhedral model

The polyhedral model is an instrument to represent the execution of a computer program in a geometric way. Its computational simplicity makes it very suitable for automatic optimization in compilers [4], [5]. Each statement of the program is characterized by its iteration domain, the access functions of written and read data, and its schedule.

Paramount for our work is that while access functions in the polyhedral model express array indices as functions of loop variables and parameters, they also map the points of the iteration domain to array elements, i.e. they define a transformation from the iteration domain to the data domain. These transformations are affine projections. The access functions can be split into two parts: the loop variables dependent part (called *motion vector*) and the constant part (called *offset*).

III. Polyhedral reuse buffer design flow

Our design flow is depicted in Fig. 1. Starting from a loop nest, we analyze array references for data reuse (section III-A). We organize array references that share data into reuse chains (section III-B), which will be mapped onto a chain of FIFO buffers. For each reuse chain we calculate the fetch domain (section III-C) and the buffer size (section III-D). Finally, an RTL design is generated (section III-E). For polyhedral calculations, we use Jolylib [6] and the Barvinok library [7].

We illustrate our flow using Sobel edge detection, an image processing algorithm that detects edges in images from the horizontal and vertical brightness gradients in a 3x3 window. Pseudocode is shown in Fig. 2. From each 3x3 window, 8 pixels are used in the calculation. Conversely, pixel values are used 8 times (or reused 7 times) in the calculations. Array references and corresponding access functions are shown in columns 1 and 2 of table I. Rows in the access functions represent index dimensions. The left two columns of the matrix represent the motion vector, the right column the offsets.

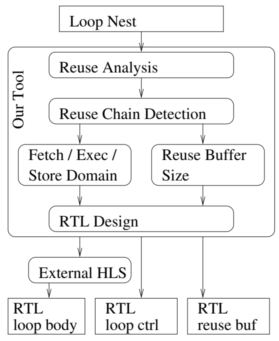


Figure 1. Polyhedral reuse buffer design flow

```
unsigned char pixel_in[cols][rows]
for (r : 1..rows-1)
  for (c : 1..cols-1)
    gradX = pixel_in[c-1,r-1] + 2*pixel_in[c-1,r]
            + pixel_in[c-1,r+1] - pixel_in[c+1,r-1]
            - 2*pixel_in[c+1,r] - pixel_in[c+1,r+1]
    gradY = pixel_in[c-1,r-1] + 2*pixel_in[c,r-1]
            + pixel_in[c+1,r-1] - pixel_in[c-1,r+1]
            - 2*pixel_in[c,r+1] - pixel_in[c+1,r+1]
    grad = abs(gradX) + abs(gradY)
    if (grad>255) grad = 255
    pixel_out[c,r] = 255 - grad
```

Figure 2. Sobel edge detection: pseudocode

Our method works on a loop nest with references to array data. For the polyhedral representation to be applicable, loop bounds and array indices need to be affine expressions of loop variables and parameters. Benchmarks studied in [8] show that these categories comprise between 83% and 100% of the array indices. Support for additional categories is future work.

A. *Reuse analysis*

Reuse of array elements may appear in 3 different ways.

1) When two array references in the loop body have the same access function. In our example, this is the case for a.o. `pixel_in[c-1,r-1]` which occurs twice.
2) When an array subscript is invariant for a certain combination of loop variables. E.g. `A[i-j]` will access the same element of `A` whenever $i-j$ attains the same value. A special case is when an access function is independent of a loop variable.
3) When two different access functions attain the same value for different combinations of loop variables. E.g. `A[i+1]` and `A[i]` will access the same array element in iterations $i=n$ and $i=n+1$, respectively.

This work focuses on cases 1 and 3, i.e. reuse between different array references. At this stage, support is limited to rectangular data domains, and array indices that depend on one loop variable only. The access functions define a transformation from the iteration domain to the array elements. For each array reference, we calculate its data domain by transforming the iteration domain of the statement in which the array reference occurs with the transformation defined by the access function. The intersection of two such data domains is the set of array elements that are accessed by both array references. In other words, reuse occurs if the data domains of different array references overlap.

Table I. ACCESS FUNCTIONS, DATA DOMAINS AND REUSE DISTANCES

Index expression	Access function	Data domain	Reuse buffer size
pixel_in[c-1][r-1]	$\begin{bmatrix}1 & 0 & -1\\0 & 1 & -1\end{bmatrix}$	$1 \le c \le cols-2$, $1 \le r \le rows-2$	
			1
pixel_in[c][r-1]	$\begin{bmatrix}1 & 0 & 0\\0 & 1 & -1\end{bmatrix}$	$2 \le c \le cols-1$, $1 \le r \le rows-2$	
			1
pixel_in[c+1][r-1]	$\begin{bmatrix}1 & 0 & 1\\0 & 1 & -1\end{bmatrix}$	$3 \le c \le cols$, $1 \le r \le rows-2$	
			$cols-2$
pixel_in[c-1][r]	$\begin{bmatrix}1 & 0 & -1\\0 & 1 & 0\end{bmatrix}$	$1 \le c \le cols-2$, $2 \le r \le rows-1$	
			2
pixel_in[c+1][r]	$\begin{bmatrix}1 & 0 & 1\\0 & 1 & 0\end{bmatrix}$	$3 \le c \le cols$, $2 \le r \le rows-1$	
			$cols-2$
pixel_in[c-1][r+1]	$\begin{bmatrix}1 & 0 & -1\\0 & 1 & 1\end{bmatrix}$	$1 \le c \le cols-2$, $3 \le r \le rows$	
			1
pixel_in[c][r+1]	$\begin{bmatrix}1 & 0 & 0\\0 & 1 & 1\end{bmatrix}$	$2 \le c \le cols-1$, $3 \le r \le rows$	
			1
pixel_in[c+1][r+1]	$\begin{bmatrix}1 & 0 & 1\\0 & 1 & 1\end{bmatrix}$	$3 \le c \le cols$, $3 \le r \le rows$	

The set of references to an array can be partitioned into subsets that have (fully or partly) overlapping data domains. We call these subsets *reuse sets*. The data domain of the reuse set is the union of the data domains of the array references in the reuse set. The third column of table I shows the data domain of each array reference in the Sobel edge example. In this case, all array accesses form a single reuse set.

B. *Reuse chain*

We now describe how to order these reuse sets into *reuse chains*. This ordering is based on the access functions. At this stage, our work is limited to reuse sets of array references that have the same motion vector, corresponding to *sliding window* access patterns, but other patterns can be added (handled as future work).

We order the array references into a reuse chain according to the time (iteration) in which they access the array elements. The array reference at the *head* of the reuse chain is the first one to access array elements. If it is a read, the array element has to be fetched from memory. The array element is then *passed* to the next references in the reuse chain where it is accessed in a later iteration, until at the end of the chain the element is no longer used. If there is a write operation in the chain, the array element needs to be written back to memory after the last write access in the chain. For the Sobel edge detection example, the rows of table I are ordered in reuse chain order from tail to head.

Reuse chains can be used to create reuse buffers. Between read accesses, or between a write and successive reads, the reuse buffer is a FIFO. Between any operation and a successive write, no buffering is needed as the existing data will be overwritten. If the head of the reuse chain is a read access, the array element needs to be fetched from memory. If there is a write operation in the chain, the array element needs to be written back to memory after the last write access in the chain.

```
for (I in extended iteration domain) {
  if (I in fetch domain)
    FETCH_ARRAY_DATA_INTO_REUSE_BUFFER(I);
  if (I in original iteration domain)
    EXEC_LOOP_BODY_WITH_REUSE_BUFFER(I);
  if (I in store domain)
    STORE_ARRAY_DATA_INTO_MAIN_MEMORY(I);
  }
```

Figure 3. Code fragment with reuse buffers

C. Fetch, execute and store domains

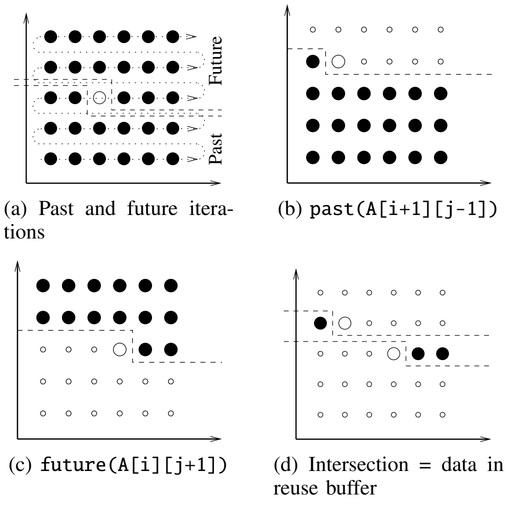
Using a reuse buffer, the loop nest looks as in Fig. 3: new data is fetched into the reuse buffer, the loop body is executed with data in the reuse buffer, and data is stored back into memory. Generally, only one new array element at the *head* of the reuse buffer needs to be fetched for each iteration. The rest is taken from the reuse buffer. Additional data fetches are needed to fetch all necessary data, e.g. to pre-fill the reuse buffer before the first execution of the loop body. We solve this by extending the iteration domain with fetch-only iterations. We call the iteration domain of all fetch operations the *fetch domain*. All fetches use the same array indices, namely those of the head of the reuse chain.

Now we calculate the fetch domain. The data domain of an array reference is the transformation of its iteration domain by the access function. Inverting these transformations is trivial as they are bijective. Transforming the data domain of a reuse chain back to the iteration domain, a new iteration domain is obtained. Choosing the inverse access function of the head of the reuse chain for back transformation, we obtain the fetch domain. Similarly, the *store domain* can be found using the array reference at the end of the reuse chain. With the original iteration domain of the loop and the fetch and store domains, we can build a new loop nest as in Fig. 3. The extended iteration domain is the union of the original iteration domain of the loop and the fetch and store domains.

In Sobel Edge, we find the data domain of the reuse chain as the union of the data domains of all accesses of array A (see table I), i.e. $1 \leq i_1 \leq cols, 1 \leq i_2 \leq rows$. The head of the reuse chain is $pixel\_in[c+1][r+1]$. The fetch domain is the inverse transformed data domain by the access function of the reuse chain head: $-1 \leq c \leq cols-1, -1 \leq r \leq rows-1$.

D. Reuse buffer size

For each pair of successive array references in the reuse chain, we calculate how many array elements the data domain contains between them. Consider a point of the iteration domain representing a *present* iteration. It is trivial to split up the iteration space into *present* (the chosen point), *past* and *future*. Fig. 4a shows a graphical interpretation. Consider two successive array references of the reuse chain, $A[index1]$ and $A[index2]$, with the latter access reusing data from the former. The reuse buffer needs to store the array elements that are accessed after the former and before the latter array access. Transforming the *past* iteration space with the first access function and intersecting the result with the data domain of the reuse chain, we find the data points that have been accessed in the past by the first array reference (Fig. 4b). Similarly, transforming the *future* iteration domain with the second access function leads to data points that will be accessed in the future by the second array reference (Fig. 4c). The intersection (Fig. 4d) is the set of data points *between* both array references. The reuse buffer needs to store one additional data element, namely the *present* element. Using this method, the buffer sizes for Sobel Edge are as shown in column 4 of table I.



(a) Past and future iterations
(b) `past(A[i+1][j-1])`
(c) `future(A[i][j+1])`
(d) Intersection = data in reuse buffer

Figure 4. Determining the size of the reuse buffer

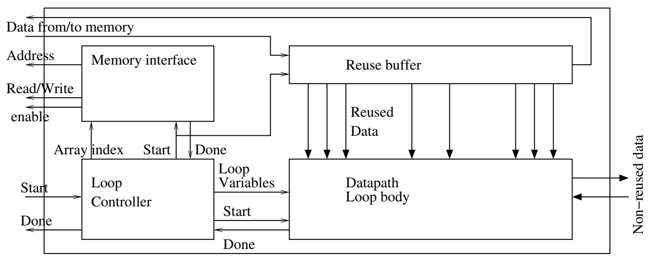


Figure 5. The overall RTL design

In case of a long reuse distance, the size of the reuse buffer may exceed available on-chip resources. In such case, the excessively long buffer should be left out, splitting the reuse chain in two separate chains and requiring an additional fetch from memory.

E. Building the RTL design

With the elements from the previous sections, we automatically generate an RTL design that exploits the data reuse. The RTL design consists of 4 parts as in Fig. 5. The *datapath* implements the data statements. It gets array data from and writes array data back to the *reuse buffer(s)*. The *loop controller* controls the execution of the loop nest, firing datapath and reuse buffer operations and calculating loop variables and array indices. The *memory interface* handles the communication between the memory and the reuse buffer(s).

The datapath can be generated with any HLS tool. To this end, our tool rewrites the loop body, replacing array references with variables that are the ports of the reuse buffer, as shown in Fig. 6. Generating the reuse buffer from the reuse chain and reuse buffer sizes is straightforward. Between each pair of accesses, a FIFO is instantiated with the appropriate length. So far our tool generates the RTL code of reuse buffers with only read operations. The loop controller generator builds on [9], with extensions for array index calculation and firing of

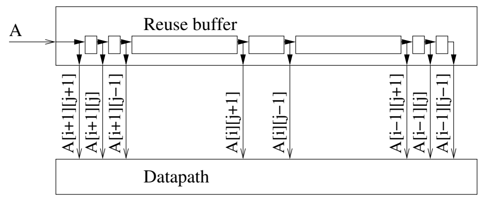


Figure 6. Reuse buffer and datapath

Table II. EXPERIMENTAL RESULTS

	Latency (cycles)	LUTs	FFs	RAM (reuse buffer)
Our tool, unoptimized code	**39,602**	**365**	**156**	**2 (8-bit wide)**
HLS, unoptimized code	105,745	290	231	none
HLS, optimized code	39,701	364	313	1 (16-bit wide)
HLS, optimized code, pipelined	10,397	417	404	1 (16-bit wide)

memory operations. The memory interface is a generic piece of RTL that asserts read / write signals as required. The VHDL code for the top level was written by hand. Correct operation of the reuse buffer was verified in simulation.

IV. EXPERIMENTAL RESULTS

We compare our approach to two alternatives that all use HLS to synthesize the Sobel Edge detector, one using the same unoptimized code as with our tool, and one, in which we have hand-optimized the source code so that HLS would produce a reuse buffer. Calypto's Catapult C was used in all cases. The image size was 100×100 pixels. The bandwidth of the input and output image buffers was 1 pixel per clock cycle, i.e. at least 10,000 cycles are needed to transfer all data. RTL synthesis was run for area estimation, targeting a Xilinx Virtex-5 FPGA. The results are in table II. The area is given in lookup table, flip-flop and RAM counts.

Without pipelining, our generated circuit greatly outperforms the circuit generated using HLS from the same source code in terms of latency, and is slightly better than the one with HLS and hand-optimized code. The area is in the same range for all circuits. The latency is still considerably worse than the theoretical optimum of slightly more than 10,000 cycles. With pipelining, which our tool doesn't do yet, HLS gets rather close to this figure. From these experiments, we conclude that our automated method performs equally well as manual optimization combined with traditional HLS. Adding pipelining to our method will further improve latency.

V. RELATED WORK

ROCCC is one of the few HLS tools that does memory access optimization. ROCCC introduces the concept of *smart buffers* [10] for input data reuse. Unfortunately ROCCC has very stringent requirements on the input C code, and the generated RTL doesn't scale well with the reuse distance [3].

Two papers discuss the generation of an application specific memory architecture similar to what we do. Ref. [11] presents methods to optimize local data storage and transfers to main memory. The authors tackle both the problem of optimizing the loop nest for the available memory resources and the generation of fitting cyclic reuse buffers through HLS. Their method for reuse buffer length calculation is more complex than ours. In [12], a method is presented to use on-chip buffers for data reuse. The authors use the polyhedral model as well as the transformation aspect of access functions. However, they don't use the reverse transformation as we do, resulting in a more complex algorithm for code generation. Their work is also limited to data reuse in consecutive iterations, which is not a limitation in our work. In both papers, the reuse buffers are introduced in C, and RTL generation is left to the HLS tool. This makes the integration of the reuse buffer and datapath easier at the expense of less control from the designer on the reuse buffer hardware design.

VI. CONCLUSION

In this paper we have presented a method to automate the generation of data reuse buffers for HLS. It uses the polyhedral model and more specifically the fact that array access functions are affine transformations between the iteration and the data space. This leads to a powerful method to analyze potential data reuse, as well as an elegant means of streamlining data fetching, processing and storing in a loop nest. Though not fully integrated yet, our toolflow is able to generate all major parts for building a functional circuit. Our automated method and flow produce circuits that perform equally well as circuits generated with HLS from manually optimized C code.

REFERENCES

[1] P. Coussy and A. Takach, "Guest editors' introduction: Raising the abstraction level of hardware design," *IEEE Design and Test of Computers*, vol. 26, no. 4, pp. 4–6, 2009.

[2] P. Coussy, D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design and Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.

[3] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Design Automation for Embedded Systems*, pp. 1–21, 2012, 10.1007/s10617-012-9096-8.

[4] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *Int. J. Parallel Program.*, vol. 34, no. 3, pp. 261–317, 2006.

[5] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam, "Putting polyhedral loop transformations to work," in *LCPC'16 International Workshop on Languages and Compilers for Parallel Computing, LNCS 2958*, College Station, October 2003, pp. 209–225.

[6] Reservoir Labs, "Jolylib," https://www.reservoir.com/.

[7] S. Verdoolaege, *barvinok: User Guide*, 0th ed., June 2007, available from http://freshmeat.net/projects/barvinok.

[8] Y. Paek, J. Hoeflinger, D. Padua, and I. C. D. Padua, "Efficient and precise array access analysis," *ACM Trans. Program. Lang. Syst*, vol. 24, p. 2002, 2000.

[9] H. Devos, K. Beyls, M. Christiaens, J. Van Campenhout, E. H. D'Hollander, and D. Stroobandt, "Finding and applying loop transformations for generating optimized FPGA implementations," *Transactions on High Performance Embedded Architectures and Compilers I, LNCS*, vol. 4050, pp. 159–178, 2007.

[10] Z. Guo, B. Buyukkurt, and W. Najjar, "Input data reuse in compiling window operations onto reconfigurable hardware," in *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*. ACM Press, July 2004, pp. 249–256.

[11] J. Cong, P. Zhang, and Y. Zou, "Optimizing memory hierarchy allocation with loop transformations for high-level synthesis," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1233–1238.

[12] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, 2013, pp. 29–38.