Optimizing hardware and software for memory and speed

Corentin Ferry – Supervision: Sanjay Rajopadhye, Louis-Noël Pouchet
Colorado State University, Fort Collins CO, USA & Univ Rennes, 35000 Rennes, France

Minimizing memory accesses and optimizing hardware resource usage to maximize accelerator performance

Minimizing memory accesses in programs

Can we give the minimal number of memory load/stores achievable by a program?

Modeling Input and Output with the Red/Blue Pebble Game

Executing a program incurs memory accesses that account for **input and output complexity** (I/O complexity). Such accesses have
- higher **latency** than other (e.g. arithmetic) operations for the data going back and forth through caches, and
- high **energy impact** resulting from the use of an external memory chip.

Hong & Kung [1] created a **red/blue pebble game** that models the input/output cost of a program, on a machine model with a single level of cache. Red pebbles model fast memory (cache), and blue pebbles model slow memory (RAM). There are **unlimited blue pebbles** and **limited red pebbles**.
The game follows four rules, each one having its own cost:
- R1 (Load): data goes from slow memory (blue) to fast memory (red),
- R2 (Store): data goes from fast memory to slow memory,
- R3 (Compute): a value is computed in fast memory from data in fast memory.
- R4 (Delete): a red pebble is freed.

A valid sequence of rules is called an **execution**.

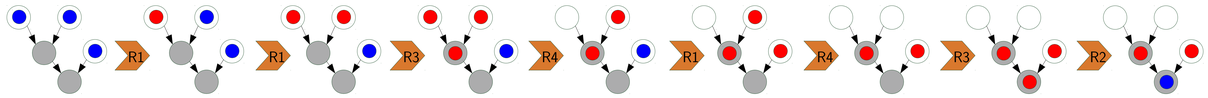


Figure 1: One possible execution of the Red/Blue Pebble Game. In white, input nodes; in gray, nodes that are computed.

Storing and fetching into memory (**spilling** and **restoring**) intermediate results is necessary when falling short of registers.

The least spills and restores a schedule incurs, the better its I/O complexity.

Computing upper bounds for small problems

I/O lower bounds are computed from a computational graph (DAG).
- For some problems such as Matrix Multiplication, the minimal I/O complexity **is exactly known** through theoretical computations.
- For arbitrary programs, the exact complexity cannot be computed with the current techniques. **Approximate lower bounds** are given [2].

We would like to compute a numerical upper bound for these bounds given a DAG. This problem is however an NP-complete problem! In an attempt to mitigate its complexity, we reduced the sets of spills and restores to be explored.

For every valid schedule S of the given DAG:
  For every subset of the values (v_i) computed in the DAG:
    Compute the actual I/O cost of the execution of S when spilling (v_i)

For every valid schedule S of the given DAG:
  Execute S until spilling is needed
  For every value v in registers/cache:
    Compute the actual I/O cost of the execution of the rest of S when spiling v

Figure 2: Optimizations of the game execution enumeration algorithm (in red)

Benchmark	Number of nodes	$\|I\|$	$\|O\|$	Schedules explored	Regs	Minimal IO found	Theor. bound	Time (s)
`matmult2`	24	8	4	5,108,103,000	5	16	3.577	6,595
				5,108,103,000	6	14	3.266	6,291
				3	7	12	3.024	1
				2	8	12	2.828	1
`fft4`	12	4	4	1,120	3	14	1.333	1
				1,120	4	12	1	1
				1,120	5	10	0.8	1
				90	6	8	0.666	1
				3	7	8	0.571	1

Table 1: Red/blue pebble game simulator results

We have proved that spilling only when it is necessary to do so, keeps an optimal execution in terms of I/O among the explored executions. We further optimized the search by **sorting schedules** according to **heuristics**, such as the increasing order of the reuse distance of the computed values.

We wrote a simulator that runs the algorithm on Figure 2. Results are in Table 1: theoretical bounds shown are asymptotic lower bounds by Hong & Kung [1]. They have been tightened in some specific cases, e.g. by Irony et al. for matrix multiplication [3].

References

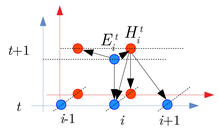
[1] Hong Jia-Wei and H. T. Kung. I/O complexity. In Proceedings of the thirteenth annual ACM symposium on Theory of computing - STOC '81. ACM Press, 1981.
[2] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. On characterizing the data access complexity of programs. ACM SIGPLAN Notices, 50(1):567–580, jan 2015.
[3] Dror Irony, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. Journal of Parallel and Distributed Computing, 64(9):1017 – 1026, 2004.
[4] Cong, J.: A new generation of c-base synthesis tool and domain-specific computing. In: 2008 IEEE International SOC Conference. pp. 386–386 (Sept 2008). https://doi.org/10.1109/SOCC.2008.4641556
[5] Irigoin, F., Triolet, R.: Supernode partitioning. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages – POPL '88. ACM Press (1988). https://doi.org/10.1145/73560.73588
[6] Puranik, M.: Optimal Design Space Exploration for FPGA-based Hardware Accelerators: A Case Study on 1-D FDTD. Master's thesis, Colorado State University (2015)
[7] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, and Robert W. Stewart. 2014. The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 all Programmable Soc. Strathclyde Academic Media, , UK.

Maximizing FPGA accelerator performance using High-Level Synthesis tools

How to automate generation of optimal architectures for stencils from equations?

Using High-Level Synthesis to write code for stencil equations

Stencils are recurrence equations with **uniform dependences**. They are usually implemented as nested loops (space, time, etc).
The dependence pattern is independent from the coordinates of a point in the iteration space.



$$H_i^t = d_a H_i^{t-1} + d_b \left(E_i^{t-1} - E_{i+1}^{t-1}\right)$$
$$E_i^t = c_a E_i^{t-1} + c_b \left(H_i^t - H_{i-1}^t\right)$$

Figure 4: FDTD-1D equation and dependence pattern

- Writing programs that compute such equations in a hardware description language (HDL) is tedious, requiring to create both the architecture and the schedule for the equation.
- High-level synthesis (HLS) tools [4] **create architectures automatically from C or C++ code**. The developer only writes the schedule, and can provide architectural hints (pragmas) to the HLS tool.

Manual Writing

```
void update_tile_left_border(D_TYPE* H, D_TYPE* E,
        volatile D_TYPE c1, volatile D_TYPE c2, volatile D_TYPE d1, volatile D_TYPE d2, ap_uint<32> i,
        D_TYPE H_left, D_TYPE H_bottom, D_TYPE E_bottom, ap_uint<1> useERight) {
#pragma HLS INLINE
    D_TYPE newH =  d1 * H_bottom + d2 * (E_bottom - ((useERight)?E[i]:0));
    H[i] = newH;
    E[i] =  c1 * E_bottom + c2 * (newH - H_left);
}
```

Figure 5: FDTD-1D equation in C++ HLS code

Automating FPGA code generation from HLS code with a custom framework

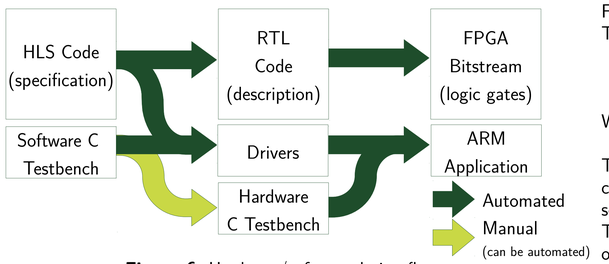


Figure 6: Hardware/software design flow

FPGA code generation is automated thanks to a custom framework we made.
Three steps are performed:
- High-level synthesis : from a C description to an hardware architecture.
- Synthesis : from an architecture description to logic gates (FPGA).
- Software : compilation of a testbench that calls the hardware.

We use Xilinx Vivado 2017.4 as a backend and HLS toolchain.

The user of our framework only needs to write the HLS description of the desired computation and a testbench. Our framework automatically links hardware and software using a shared space in memory.
The whole flow in steady state takes from **5 minutes to 35 minutes** depending on the design size. The developer can quickly assess the design performance and make design choices to improve it.

Optimizing HLS code using the polyhedral model

We can use the polyhedral model to optimize stencils for FPGAs. This mathematical framework provides several code transformations, including tiling [5], skewing, loop permuting, loop fusing. Puranik [6] applied these transformations, along with a clever manual optimization to the FDTD-1D stencil.

We took Puranik's schedule [6] and tiling to write code for our HLS framework:
- The iteration space is chopped into tiles. Each processor is assigned one tile. Tiles executed in parallel are grouped into passes.
- Passes are run consecutively, eliminating a need for inter-pass synchronization.

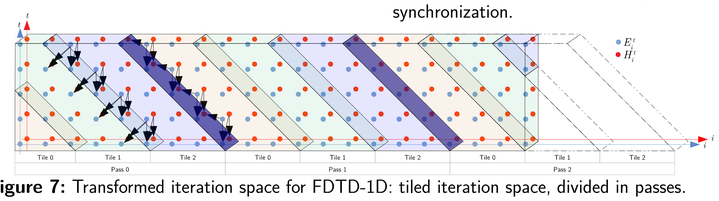


Figure 7: Transformed iteration space for FDTD-1D: tiled iteration space, divided in passes.
Arrows show inter-pass dependences (implemented as buffer) and inter-tile dependences (registers).

Exploring the design space: Looking for the peak performance

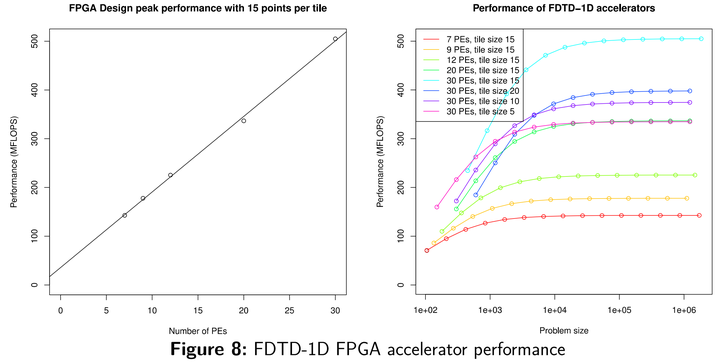


Figure 8: FDTD-1D FPGA accelerator performance
(left: vs. number of processors at fixed problem size, right: vs. problem size)

- We used a Xilinx Zynq-7000 FPGA (XC7Z020-1CLG400) with SoC running the software part.[7]
- Our results show a linear performance improvement with the number of processors (PEs). This is compatible with Puranik's analytical model [6].
- Maximal performance delivered: 500 MFLOPS for problem sizes over 100.000 points with 30 PEs, 15 points per tile × 10 iterations.
  This result can be improved by telling HLS that inter-iterations dependences are always honored.

Going further, we would like accelerator generation in HLS to be automated with tools like PoCC, Pluto (source-to-source polyhedral compiler), and AlphaZ (translates equations into C code).