Application-Specific Memory Access by Prefetching via High Level Synthesis

İsmail San and Atakan Doğan

Dept. of Electrical and Electronics Engineering
Faculty of Engineering
Anadolu University
Eskişehir, Turkey
Email: {isan, atdogan} @anadolu.edu.tr

Kemal Ebcioğlu

Global Supercomputing Corporation
Yorktown Heights, NY, USA
Email: kemal.ebcioglu@global-supercomputing.com

Abstract—**High Level Synthesis (HLS) allows an automatic translation from high level C/C++ descriptions into Register Transfer Level (RTL) hardware designs. HLS enables to design at a high level of abstraction that offers one to focus on high level concepts within less amount of design time. Once a specific data intensive application is considered to be accelerated in hardware, its memory access pattern must be exploited for higher performance. Most of the time, an application suffers from a high amount of memory access latencies. To reduce the memory access latencies, we use widely known prefetching technique to mask the latencies. In this paper, we enabled a data prefetching scheme specified at the C/C++ descriptions level via Vivado$^{\text{TM}}$HLS, which overlaps double-buffered prefetching and computation.**

Keywords–Prefetching; High level synthesis; Memory access time; Off-chip memory latency; Application-specific memory hierarchy.

I. Introduction

Modern VLSI technology allows Integrated Circuit (IC) manufacturers to build a single IC chip consisting of billions of transistors [1]. This advancement in VLSI technology requires complex Computer-Aided Design (CAD) tools to place and route billions of transistors. Advanced CAD tools must deal with this complexity and offer an acceptable design time. High Level Synthesis (HLS) tools allow hardware and software designers to create Register Transfer Level (RTL) hardware architectures from C/C++ high level descriptions [2]–[4]. In turn, current RTL synthesizers convert RTL specifications into Field-Programmable Gate Array (FPGA) bitstreams or IC chips. When performance is critical for the applications, designing application-specific hardware accelerators becomes inevitable. FPGAs offer reconfigurability for implementing arbitrary digital circuits: one can exploit all the parallelism in the algorithm to get more performance. FPGAs are used in many different fields from networking to data centers. Currently, they can also be accessed in the cloud. One can run customizable Xilinx FPGAs in the Amazon Elastic Compute Cloud (Amazon EC2) F1 instances [5].

When we consider a specific application that needs to be accelerated in hardware, it is of utmost importance to consider how the application is accessing the memory, since memory access time limits the peak performance of many data intensive applications. To reduce the memory access time, the memory access pattern of the application is utilized by designing application-specific memory hierarchies. Attaching caches in between memory and the processor, or hardware and software prefetching helps to reduce the memory access penalties. Since HLS generates an application-specific hardware accelerator from a high level description, it is very important to exploit the static memory references pattern inherent in the algorithm to be accelerated by the HLS. For this purpose, we questioned whether today's HLS tools generate prefetching hardware from C/C++ high level description. The key idea of our methodology is to take advantage of the fixed static memory pattern which is inherent in the algorithms, by requesting the data, which will be referenced soon, beforehand.

The rest of the article is organized as follows: Section II overviews the related work on caches and prefetching. Our motivating example for prefetching via Vivado$^{\text{TM}}$ is given in Section III. Design philosophy is described in Section IV. We discuss our implementation results on a Xilinx FPGA device in Section V and conclude in Section VI.

II. Related Work

Microarchitectural optimizations have been utilized to improve the on-chip cache bandwidth and hit rate for a given set of applications in state of the art methods on FPGA memory hierarchies. Caches exploit temporal and spatial locality by keeping recently used data in the cache and also bringing the nearby data into the cache, respectively. General memory hierarchies use the locality in the memory references of the algorithms via caches. In fact, caches eliminate several memory accesses, however, they will not eliminate the memory latency [6]. When a referenced data at a given address is not in the cache, a cache requests the data from the main memory and causes the processor to wait until the requested data arrives at the cache. Caches are efficient ways of connecting a general purpose processor to a memory.

In [7], the authors automatically construct a multi-cache architecture by automating the cache sizing to achieve higher cache bandwidth and hit rate. In contrast to [7], the method here presented focuses on fetching data before they are used in the execution in order to hide the memory access latency.

Prefetching is a known technique that accesses and stores data into a temporary buffer before it is needed; it aims to hide memory latencies. Applications having regular memory accesses, which are deterministically known prior to the execution, can take advantage of prefetching whereas the other applications with random memory accesses cannot utilize prefetching optimizations. Hardware-based data prefetching reduces the processor stall time by fetching data into the local memory before its use in the processor [6].

Recent work has used two data preloading techniques as prefetching and access/execute decoupling for accelerator-based systems [8]. The framework adds tags to accelerator memory accesses so that hardware prefetching can effectively preload data for accesses with regular patterns. On the other hand, our work does not include any tag inclusion for memory requests. Our technique employs the double-buffered prefetching and computation.

A software prefetch mechanism, which exploits the access pattern of multimedia and image processing applications, by using the DMA mode is proposed in [9] to improve the performance and reduce the overall power consumption. In contrast to [9], the present study focuses on high level synthesis of an application-specific hardware architecture with a data prefetch unit.

Automatic insertion of application-specific prefetching units is valuable in terms of hiding memory access latencies [10]. As stated in [10], automated synthesis of prefetching units can be enabled if the information is provided about when data is available for prefetching and when it is used by the application. Automatic synthesis of application-specific prefetching units, which fetch data from off-chip memory and store it in the on-chip caches in advance, is also proposed as a future work in [11]. LEAP scratchpads [12] are extended to automate the construction of application-specific memory hierarchies [11]. As emphasized by Winterstein *et al.* [11], “Knowledge about access patterns will also be used to implement application-specific prefetching and request merging”. With exploiting access patterns that is inherent in the algorithm, application-specific prefetching is a key to improve the performance of memory intensive applications.

The key element of our approach is to take advantage of the static memory access pattern at high level, which is inherent in the algorithm, by requesting the data in advance. The prefetching technique, which is shown by the PMM algorithm (Algorithm 2), uses a manual fusing of two inner loops and double-buffered prefetching, so that a few memory latencies are eliminated.

III. Motivating Example

Our motivation relies on the fact that data that will be referenced in the future can be prefetched into a buffer on the chip before they are used in a computation, which is widely known in the area as a hardware prefetching technique. We investigate whether such a hardware prefetching technique can be enabled at a high level of abstraction and whether the corresponding RTL design can be generated using an HLS tool. For our experiments, we use the Vivado™ HLS tool version 2017.4.

Let us consider matrix multiplication, since it is employed in many different fields of study from control theory to machine learning algorithms. Matrix multiplication involves several load operations that can be prefetched before they are used in the multiply and add execution unit. Matrix multiplication is defined as

$$\mathbf{C} = \mathbf{A} \times \mathbf{B}$$

where $\mathbf{A} \in \mathbb{Z}^{M\times K}$, $\mathbf{B} \in \mathbb{Z}^{K\times N}$ and $\mathbf{C} \in \mathbb{Z}^{M\times N}$. In order to compute $\mathbf{C}$, Equation 1 is used.

$$c_{ij} = \sum_{k=1}^{K} a_{ik} b_{kj} \tag{1}$$

where $1 \leq i \leq M$ and $1 \leq j \leq N$.

Once it is infeasible to store the entire input and output matrices within on-chip memories, one should store the matrices in main memory (which usually is outside the chip), fetch the data into the chip, do the computation and store back the result into the off-chip main memory.

While designing application-specific hardware, prefetching data from memory instead of caching provides more performance since a prefetching design avoids cache misses. Cache misses occur once referenced data is not in the cache memory, but, in a prefetching technique, the data is usually brought to be near the computation so that the computation never waits for the data.

While accessing the memory via caches, there is a miss penalty. The miss penalty decreases the peak performance. With perfect prefetching that masks all the memory latencies within the application, there is no miss penalty, when compared to matrix multiplication with caches. Caches exploit the locality information inside consecutive memory references. Instead of such a general solution that is provided by caches, we exploit application-specific memory patterns and generate memory references to data before the data is needed by computation. We propose a prefetching mechanism that is implemented via HLS, for a matrix multiplication operation. This prefetching mechanism can also be applied to other application-specific hardware designs if data reuse exists and data references are static.

IV. Methodology

Algorithm 1 shows the pseudo code which is very close to C/C++ description for the matrix multiplication that we choose as a reference. Translating this C/C++ description using

Algorithm 1 RMM: Ordinary Matrix Multiplication

```
Input: $A_{n\times n}$ and $B_{n\times n}$ matrices
Output: $C_{n\times n}$ matrix
1.  for $i = 0$ to $n-1$ do
2.      for $j = 0$ to $n-1$ do
3.          $S \leftarrow 0$
4.          for $k = 0$ to $n-1$ do
5.              $S \leftarrow S + A[j][k] * B[k][i]$;
6.          end for
7.          $C[j][i] \leftarrow S$;
8.      end for
9.  end for
10. return $C$
```

Vivado^{TM} HLS, generated hardware accelerator accesses to the main memory through three Advanced eXtensible Interface (AXI) interfaces for its three matrices. In main memory, three matrix addresses are not overlapped and referenced with different offset values. Our RMM algorithm, which we use as a reference, loads values of the input matrices from memory, does the computation, stores the result back to the main memory and continues with the next iteration. There is always a memory latency for each reference to data; this latency is the minimum time interval between the point in time a data is requested and the point in time the data is used in a computation.

Algorithm 2 describes our proposed prefetched matrix multiplication method. While one of the matrix row (column) buffers is used for the computation, the other matrix row (column) buffer is filled up with the next row (column) of the matrices that will be used next. This kind of double-buffered prefetching avoids stalls. As a result of converting the imperfect loop nest to a perfect loop nest, the HLS tool flattens the loop nest and saves extra cycles that would otherwise be spent when entering or exiting the loops [13]. A perfect loop nest refers to a nested loop where only the innermost loop has a body. Thus, in Algorithm 2, we have three nested loops and only the innermost loop has the body.

Algorithm 2 PMM: Prefetched Matrix Multiplication

```
Input: $A_{n\times n}$ and $B_{n\times n}$ matrices, $CB$ and $C2B$ refers to the
    column prefetch buffers, $RB$ and $R2B$ refers to the row
    prefetch buffers. $S$ stands for the accumulated sum.
Output: $C_{n\times n}$ matrix
1.  for $r = 0$ to $n-1$ do
2.    $CB[r] \leftarrow A[r][0]$
3.    $RB[r] \leftarrow B[0][r]$
4.  end for
5.  for $i = 0$ to $n-1$ do
6.    for $j = 0$ to $n-1$ do
7.      for $k = 0$ to $n-1$ do
8.        if $k == 0$ then
9.          $S \leftarrow 0;$
10.       end if
11.       $S+ \leftarrow ((j\%2 == 0)\ ?\ RB[k]\ :\ R2B[k]) *$
              $((i\%2 == 0)\ ?\ CB[k]\ :\ C2B[k]);$
12.       if $j\%2 == 0$ then
13.         $R2B[k] \leftarrow A[(j+1)\%n][k];$
14.       else
15.         $RB[k] \leftarrow A[(j+1)\%n][k];$
16.       end if
17.       if $i\%2 == 0$ && $k == 0$ then
18.         $C2B[j] \leftarrow B[j][i+1];$
19.       else if $i\%2 == 1$ && $k == 0$ then
20.         $CB[j] \leftarrow B[j][i+1];$
21.       end if
22.       if $k == (n-1)$ then
23.         $C[j][i] \leftarrow S;$
24.       end if
25.     end for
26.   end for
27. end for
28. return $C$
```

V. Results

We described the RMM and PMM algorithms in C language and generated the hardware designs using Vivado^{TM} HLS tool version 2017.4. We also prototyped the RMM and PMM accelerators on the Nexys 4 Double Data Rate (DDR) FPGA board which contains an Artix-7 FPGA and measured the actual timings on the FPGA board. We designed a system-on-chip (SoC) architecture (see Figure 1) to verify and measure the latencies on FPGA.

Figure 1 illustrates the block design of the system-on-chip used for our experiments on the FPGA. The MicroBlaze processor is employed in this SoC to configure all the peripherals and measure the elapsed time of the RMM and PMM accelerators. The design contains a timer peripheral to measure the elapsed time and a Universal Asynchronous Receiver-Transmitter (UART) peripheral to see the data and the values taken from the timer. The Nexys4 DDR FPGA board contains a Micron MT47H64M16HR-25:H DDR2 memory component and can be accessed through a DDR controller provided by Vivado^{TM} design tool. We deployed this peripheral in our SoC and accessed it through an AXI interconnect.

In order to verify whether the generated PMM hardware does prefetching, we set up an experiment on the Nexys 4 DDR FPGA board and monitored the DDR2 slave port to check the incoming read and write addresses and their order. Figure 2 shows the waveform of the DDR2 S_AXI signals including s_axi_araddr (slave read address) and s_axi_awaddr (slave write address). One can see that the first result (c_{00}) of the matrix product is available at the 2048th cycle, which we marked on the Figure 2. The beginning address of the matrix B, A and C is 0x80000000, 0x80100000 and 0x80200000, respectively. Slave write address at this point is 0x0200000, which is the address of c_{00}, as exactly expected. The address of the beginning element of the second row, a_{10}, is 0x0100100. As one can observe that the 0x0100100 address is being fetched before the write operation of the first result c_{00}. It means that while execution stage is computing the c_{00} element, it starts to prefetch the second row of one of the input matrices which will be used in the next j iteration. Here, only one element of the second column of the other input matix is prefecthed on every j iteration.

Figure 3 shows the waveform of the same DDR2 signals as in Figure 2. Now, the first result (c_{00}) of the matrix product is available at the 2047th cycle which we marked on the Figure 3 with a marker. Slave write address at this point is 0x0200000 which again indicates the address of c_{00}. The address of the beginning element of the second row, a_{10}, is again 0x0100100. In this case, 0x0100100 address is being referenced after the first computation is finished. So, there is no prefetching in this RMM design.

We take advantage of the intrinsic memory access pattern of a specific application to decrease the total memory access latencies by overlapping double-buffered prefetching with computation. Thanks to this approach, we implemented prefetching via the HLS tool and achieved a smaller amount of total main memory access time as compared to an an algorithm without prefetching, as listed in Tables I and II. A careful scheduling technique in our prefetched matrix multiplication algorithm also helps us to totally avoid memory access stalls. As a

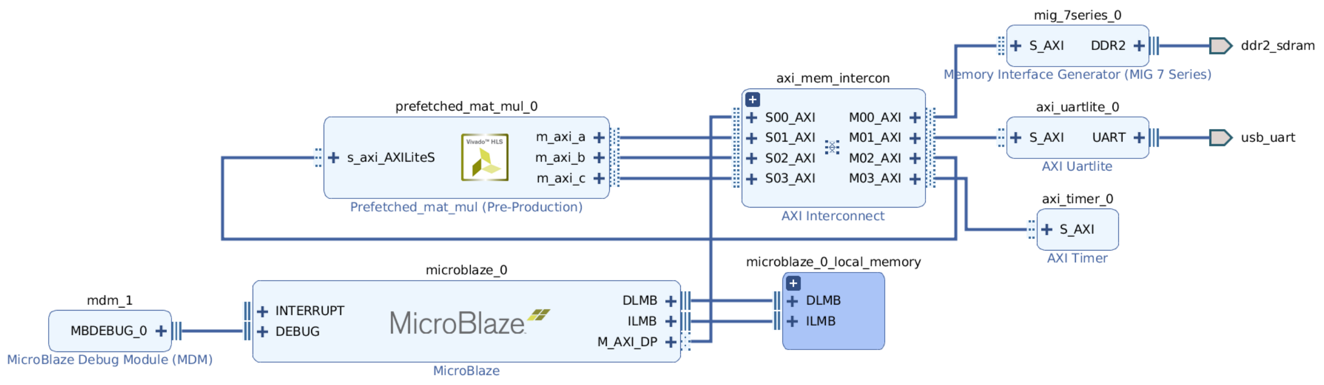
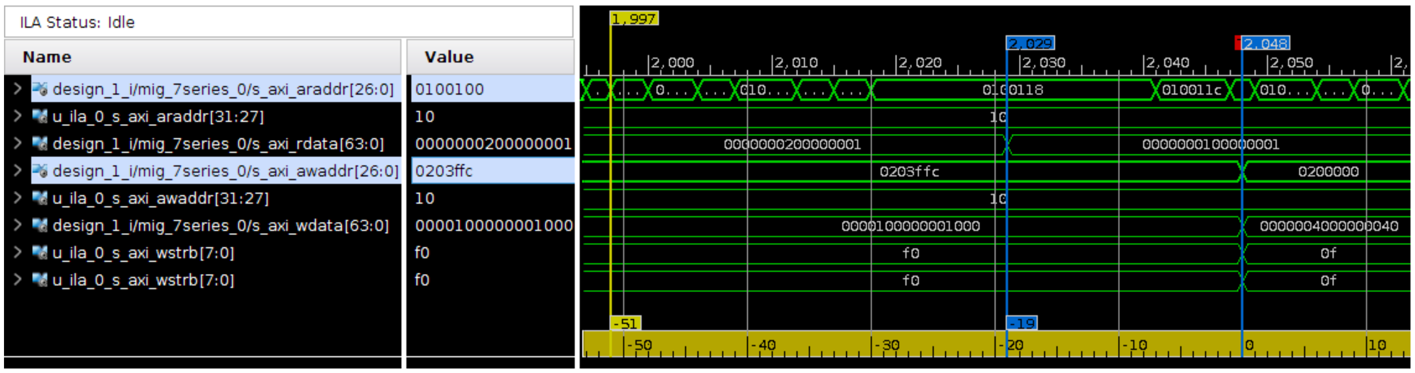


Figure 1. Block design view captured from Vivado$^{\text{TM}}$ Design Suite 2017.4 version. MicroBlaze is the processor where we configure and control all the peripherals including DDR2 controller, Timer and UART.


ILA Status: Idle

Name	Value
design_1_i/mig_7series_0/s_axi_araddr[26:0]	0100100
u_ila_0_s_axi_araddr[31:27]	10
design_1_i/mig_7series_0/s_axi_rdata[63:0]	0000000200000001
design_1_i/mig_7series_0/s_axi_awaddr[26:0]	0203ffc
u_ila_0_s_axi_awaddr[31:27]	10
design_1_i/mig_7series_0/s_axi_wdata[63:0]	0000100000001000
u_ila_0_s_axi_wstrb[7:0]	f0
u_ila_0_s_axi_wstrb[7:0]	f0

Figure 2. A window of the timing diagram for the PMM captured from Chipscope tool of the Vivado$^{\text{TM}}$ design suite 2017.4. Slave port of the DDR2 component signals are captured in this diagram.

TABLE I. RESULTS OF PMM AND RMM DESIGNS ON ARTIX-7 FPGAS. THE VALUES ARE LATENCY ESTIMATES GATHERED FROM VIVADO$^{\text{TM}}$ HLS 2017.4.

Design	Size	BRAM_18K	DSP48E	FF	LUT	Latency [cycles]
RMM	16	6	3	1908	2311	10497
PMM		10	3	3108	4359	4325
RMM	32	6	3	1924	2319	58369
PMM		10	3	3126	4384	33013
RMM	64	6	3	1940	2327	364545
PMM		10	3	3144	4398	262421
RMM	128	6	3	1956	2341	2506753
PMM		10	3	3162	4416	2097493
RMM	256	6	3	1972	2357	18415617
PMM		10	3	3180	4444	16777685

†All designs are targeted to 10 ns clock period.

consequence, the proposed prefetching method is achieving higher performance thanks to the HLS tool.

Table I summarizes the utilization and performance results taken from Vivado$^{\text{TM}}$ HLS 2017.4 for PMM and RMM hardware designs with different matrix sizes. For instance, when we consider 16×16 matrices for the matrix multiplication, PMM-16 design takes 4325 cycles to complete the operation which is better than the RMM-16 in terms of latency. Since we use block memories (BRAM_18K) to implement four buffers in the PMM, it requires 4 more buffers than the PMM for all sizes. Approximately, 1.5 times more Flip-Flop (FF) and 2 times more Lookup Table (LUT) is required in the PMM. Once application performance is critical (which is often the case in the hardware accelerators), prefetching provides more performance, but with an area overhead.

Table II lists the performance estimates and the actual hardware results in terms of cycles of PMM and RMM with matrix size of 64. As one can see, the PMM-64 is completing the matrix multiplication approximately 1.6 times faster than the RMM-64. Due to the latencies in the AXI interconnect and the DDR2 memory, actual hardware measurements are higher than the Software (SW) estimates. The software estimates are measured without considering the latencies caused by the accelerator interfaces.

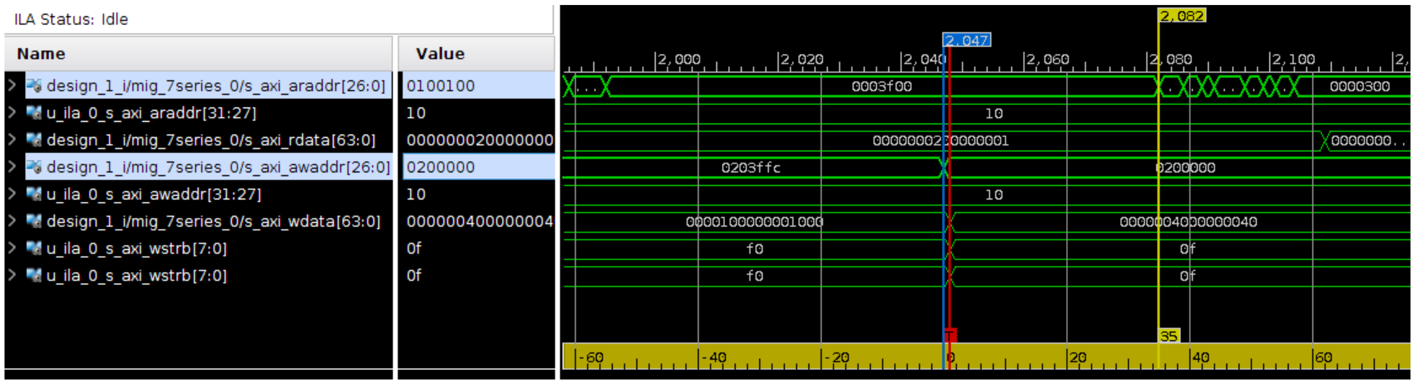


Figure 3. A window of the timing diagram for the RMM captured from Chipscope tool of the Vivado$^{\text{TM}}$ design suite 2017.4. Slave port of the DDR2 component signals are captured in this diagram.

TABLE II. RESULTS OF PMM AND RMM DESIGNS ON ARTIX-7 FPGAS. THE VALUES ARE GATHERED FROM FPGA IMPLEMENTATION.

Design	SW Estimates [cycles]	HW Results [cycles]
RMM-64	364545	3188847
PMM-64	262421	1960804

†All designs are targeted to 10 ns clock period.

VI. Conclusion

HLS provides an improved design methodology because it reduces design time and because it allows specifying an algorithm at a high level. Design space explorations becomes prominent and applying high level concepts without touching the difficult low level abstraction layer is possible. Memory bandwidth is the bottleneck that limits the performance of many different data intensive applications. In order to decrease the memory latencies, we enabled a prefetching mechanism using the HLS tool and obtained results with a case study of the matrix multiplication algorithm. The main advantage of prefetching is that one can mask all the memory latencies if computation is taking more time than total memory access time and if dependences permit. If the memory access pattern of an application is static, prefetching is an efficient way to improve performance and can be described by Vivado$^{\text{TM}}$ HLS.

Note that while currently the Vivado$^{\text{TM}}$ HLS tool cannot automatically add prefetching to a C algorithm as we have accomplished in the present paper, we are not claiming that an HLS compiler in general cannot automatically create an application-specific hardware with prefetching from a C algorithm. For example, a compiler technique called *loop fusion* [14] applied to two loops in sequential C code at an intermediate step of possible compiler transformations starting from the original RMM algorithm and ending with the PMM algorithm:

1) loop to do computation on the current row (column) buffer
2) loop to do prefetching to fill the next row (column) buffer

can result in a single loop where the the next row (column) is prefetched into the next row (column) buffer while computation proceeds at the same time on the current row (column) buffer.

However, one compiler challenge in achieving perfect prefetching with loop fusion is to *fuse loops of unequal shapes*, which is in effect used by the prefetching technique of the present paper: consider a loop (1) to perform computations on the current column buffer which is a doubly nested loop (the j loop, which includes the k loop as an inner loop) and consider a loop (2) for filling the next column buffer which is an ordinary singly nested loop. Loops (1) and (2) must be fused to accomplish effective prefetching. In the present PMM algorithm a manual fusing has been done by moving an incremental prefetching of 1/n'th of the next column of B, all the way into the inner k loop, which is an example of fusion of loops with unequal shapes.

Further work is needed to automatically design an application-specific perfect prefetching mechanism through HLS in order to fully mask memory latency time (i.e., so that referenced data is always in the SRAM of the processor). A worthwhile research direction is to schedule and software pipeline a general sequential code loop nest, and to generate parallel hardware and a memory hierarchy from the code such that each memory operand is already in a register or SRAM location when it is needed for computation, whenever dependences permit [15].

References

[1] S. K. Kundu, S. Karmakar, G. S. Taki, A. Roy, C. D. Choudhuri, M. Basu, A. Basak, R. Upadhyay, A. Raj, and S. Mandal, "Progress in submicron device technology," in 2017 8th Annual Industrial Automation and Electromechanical Engineering Conference (IEMECON), Aug 2017, pp. 318–320.

[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 30, no. 4, April 2011, pp. 473–491.

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis

for FPGA-based processor/accelerator systems," in Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays. ACM, 2011, pp. 33–36.

[4] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 10, Oct 2016, pp. 1591–1604.

[5] "AWS FPGA Developer AMI," https://aws.amazon.com/marketplace/pp/B06VVYBLZZ, Accessed: 2018-04-04.

[6] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," IEEE Transactions on Computers, vol. 44, no. 5, May 1995, pp. 609–623.

[7] F. Winterstein, K. Fleming, H.-J. Yang, J. Wickerson, and G. Constantinides, "Custom-sized caches in application-specific memory hierarchies," in Field Programmable Technology (FPT), 2015 International Conference on. IEEE, 2015, pp. 144–151.

[8] T. Chen and G. E. Suh, "Efficient Data Supply for Hardware Accelerators with Prefetching and Access/Execute Decoupling," in The 49th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO-49, 2016, pp. 46:1–46:12.

[9] M. Dasygenis, E. Brockmeyer, B. Durinck, F. Catthoor, D. Soudris, and A. Thanailakis, "A Combined DMA and Application-specific Prefetching Approach for Tackling the Memory Latency Bottleneck," IEEE Trans. Very Large Scale Integr. Syst., vol. 14, no. 3, Mar. 2006, pp. 279–291.

[10] F. J. Winterstein, S. R. Bayliss, and G. A. Constantinides, "Separation Logic for High-Level Synthesis," ACM Trans. Reconfigurable Technol. Syst., vol. 9, no. 2, Dec. 2015, pp. 10:1–10:23.

[11] F. Winterstein, K. Fleming, H.-J. Yang, S. Bayliss, and G. Constantinides, "MATCHUP: Memory Abstractions for Heap Manipulating Programs," in Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '15, 2015, pp. 136–145.

[12] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer, "Leap scratchpads: Automatic memory and cache management for reconfigurable logic," in Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, ser. FPGA '11, 2011, pp. 25–28.

[13] "Vivado Design Suite User Guide - High-Level Synthesis," 2017. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug902-vivado-high-level-synthesis.pdf

[14] Wikipedia, "Loop fission and fusion," 2018, [Online; accessed 8-June-2018]. [Online]. Available: https://en.wikipedia.org/wiki/Loop_fission_and_fusion

[15] K. Ebcioglu, E. Kultursay, and M. T. Kandemir, "Method and system for converting a single-threaded software program into an application-specific supercomputer," Feb. 24 2015, US Patent 8,966,457.