Analytical Model for Memory-Centric High Level Synthesis-Generated Applications

Maria Angélica Dávila-Guzmán*, Rubén Gran Tejero†, María Villarroya-Gaudó‡ and Darío Suárez Gracia§
gaZ-DIIS-I3A, Universidad de Zaragoza — HiPEAC Network of Excellence
e-mail: *angelicadg@unizar.es, †rgran@unizar.es, ‡mvg@unizar.es, §dario@unizar.es

Abstract—High performance computing (HPC) demands huge memory bandwidth and computing resources to achieve maximum performance and energy efficiency. FPGAs can provide both, and with the help of High Level Synthesis, those HPC applications can be easily written in high level languages. However, the optimization process remains time-consuming, especially when based on trial-and-error bitstream generation. Model-based performance prediction is a practical and fast approach for kernel optimization, specially if done with information from pre-synthesis reports. This article presents an analytical model focused on memory intensive applications that captures the memory behavior and accurately predicts the kernel execution time within seconds rather than hours, as bitstream generation requires. The model has been validated with two DRAM technologies: DDR4 and HBM2, with a set of microbenchmarks and high performance computing applications showing an average error of 11% for DDR4 and 10% for HBM2. Compared with previous studies, our predictions at least halve the estimation error.

Index Terms—Analytical model, FPGA, HLS, DRAM, HBM2, OpenCL.

1 Introduction

In the race to improve computer performance and energy efficiency beyond Moores's law in high performance computing (HPC) applications, there are new opportunities for field-programmable gate arrays (FPGAs) in a heterogeneous system. The main advantage of FPGAs comes from dedicated hardware that provides high pipeline parallelism at low power consumption.

Besides using parallelism, many HPC applications exceed the available memory bandwidth. Although external memory on FPGAs is evolving from external DRAM to 3D-stacked high bandwidth memory (HBM) increasing bandwidth from 25 to 409 GB/s, the performance of memory individual bank has a slow growth data rate of 7% per year. Comparing this with FPGA resources that grow capacity at 48% per year [1], the memory wall problem in FPGA applications is evident.

Exploiting the potential benefits of FPGA technology is a challenge for programmers, even with the development of high level design with languages such as C, C++, or OpenCL [2]–[4]. Generating highly tuned code is time consuming, and CPU or GPU optimization techniques are not always suitable for FPGAs. Programmers have two options for easy coding. Either they write well-known code patterns from previous explorations [5], [6], or they rely on pre-synthesis analytical models for estimating performance [4], [7]–[9]. These models analyze the RTL code from High-level synthesis (HLS) compiler tools, the high-level code, or both.

Model-based optimization seeking to better exploit FPGA resources and simplify hardware generation focuses mainly on the compute part, or kernel pipeline. Previous studies have oversimplified the organization of global memory interconnect (GMI), which manages memory request between the kernel-pipeline and the off-chip DRAM memory. The lack of detail in the models results in the error seen in two state-of-the-art analytical models [8], [9]. It multiplies by 3 when the DRAM specification changes and can be larger than 50% for accesses with data dependencies since those models ignore the differences in memory access and DRAM technologies. Such errors could become more common in future systems because of technological advances to high-memory bandwidth devices.

This paper analyzes the GMI and their interaction with the external memory compiled with HLS tools, which affect the effective memory bandwidth and can significantly impact execution time. Combining information from the analysis of the GMI an their main components, such as load-store units (LSUs), plus the DRAM organizations, enables us to build an analytical model to accurately estimate the execution time. The model mainly requires static information from pre-synthesis reports, Verilog hardware instances, and DRAM memory timing parameters.

The contributions of this study are: a) a detailed description of the GMI for HLS, b) the first, to the best of our knowledge, analytical model that estimates the execution time of HLS-compiled memory intensive applications, c) a novel classification of the FPGA kernel-pipeline state based on memory bandwidth use, and d) a set of hints derived from observations of the analytical model to identify bottlenecks and guide programmers for optimizing kernels.

The rest of the paper is organized as follows. Section 2 presents a motivation example, Section 3 discusses state-of-the-art, Section 4 presents the HLS flow, GMI architecture and performance estimation in FPGAs. Section 5 introduces the model. Section 6 summarizes the methodology. Section 7 validates the model, and Section 8 sets out our conclusions.

2 Motivation

Optimizing HLS code often faces the challenges of establishing the kernel performance limits and determining how close to these limits an implementation is. For example, the

padding (PAD) is an algorithm that fills a matrix with an extra column of zeros; it was intensively explored in Chai [10] and optimized in Boyi [11] with 37 OpenCL design combinations where the best possible kernel achieves a 198 $\times$ speedup relative to the worst implementation.

PAD is an example of why centering the optimizations only in the OpenCL execution model can give under-optimized results. For example, our model is able to to find a memory performance bottleneck in the best Boyi's implementation. The bottleneck is the two continuous access, both with a LSU request memory width (ls_width) of 4 bytes that do not saturate memory-bandwidth and limit performance. That is to say, PAD is bounded by the low memory width, which under-uses the memory burst length. In fact, the kernel mainly performs conditional memory accesses.

With the hints from the analysis, a programmer can restructure the kernel, as Listing 1 shows, by (1) zeroing the output buffer to remove conditionals of line 4 and (2) unrolling loops to increase coalescence and the memory width to a value of 64 bytes (line 14). These modifications doubles performance running on a Stratix 10 GX FPGA. Besides, the model predicts this improvement with an estimation error of 4.6%. As a result, the memory bandwidth used in the two LSUs of the baseline and optimized PAD algorithm reaches a gain of $2.7\times$ in this work.

```
1  /** Baseline padding matrix(m x n) **/
2  int size =(m * (n + pad))
3  for(int i = size-1; i >= 0;--i){
4      if((i % (n + pad)) < n)
5          matrix[i]=read_channel_intel(channel);
6      else matrix[i] = 0.0f;  }
7
8  /** Optimized Version **/
9  float16 tempch;//increase coalescence
10 float *pch = &tempch;
11 for(int i  = 0; i < m ; i++) {
12     for(int j  = 0; j < n ; j+=V) {
13         tempch = read_channel_intel(channel);
14         #pragma unroll 16 //Unpack
15         for (int k = 0; k < 16 ; k++)
16              matrix[i*n + j*16 + k + pad_ind]=pch[k];
17     }pad_ind+=pad; //adds pad to index
18 }
```

Listing 1. Code snippets of the baseline (lines 1-6) and optimized (lines 8-18) versions of PAD.

The model helps to infer the main kernel limitations and suggests approaches for continuing the optimization process as in this case, potentially reducing the costly number of compilations, as previous works do.

3 State of the Art

Performance modeling of FPGAs using HLS has attracted the attention of many researchers to ease kernel optimizations. The standard tools from the two main FPGA vendors, Xilinx and Intel, help to address optimizations with analytical reports. In Intel case, tools focus only on three performance metrics: Initiation interval, latency, and frequency without execution time estimations.

Existing performance models target one of two different domains: embedded FPGAs [12]–[14], usually using C/C++ as the high-level language, or HPC discrete FPGAs with external components such as DRAM memories and PCIe ports. In the latter case, the models are mainly oriented to OpenCL codes [4], [8], [11] and C/C++ [9]. While embedded and HPC models have similarities in the pipeline model, the key difference is the memory system, where the throughput of an internal memory may be 380 $\times$ better than an external one.

In HPC, the memory wall is one of the main limitations of FPGAs for applications. The memory requires a controller to reorder requests to minimize row conflicts, and as a consequence the throughput depends on memory controller implementation [15], [16]. The behaviour of memory controllers is often overlooked [17], [18] or simplified as in the performance model proposed by Wang et al. [8] for Intel OpenCL SDK. It uses a coarse grain model which shows inaccuracies in the memory estimation and requires the extraction of LLVM-IR information that is not provided by the vendor's compiler. In a similar way, the Boyi framework [11] limits the memory estimation to sequential or random accesses with fixed weights, and although this framework is mainly based on memory access optimization, it only evaluates how the OpenCL execution model changes accesses.

Coarse-grained memory models reduce the optimization capabilities on HLS for FPGAs, this problem being detected by the FlexCL framework [4] for Xilinx FPGAs. FlexCL improves models covering memory access patterns with a short CPU/GPU execution, but it continues being the main source of error of the model. As some comparisons show, the memory controller makes differences in the access pattern and hence performance [15], [19]–[21]; moreover, CPU/GPU devices have a more sophisticated memory hierarchy that can hide DRAM latency. As well as the memory controller, the memory standard or technology changes the interaction with the FPGA pipeline. For this reason, the inclusion of memory parameters to cover these technology differences is necessary, DRAM technology being the most widely used. Approaches such as FlexCL obtain latency parameters from modeling the memory latency [9] as HLScope+ does [22]. Other performance estimators for Xilinx FPGAs include physical DRAM specifications, but these limit access patterns to sequential and random, as a result of their platform experiments; also, HLScope+ includes a correction factor given the lack of knowledge about the Xilinx DRAM controller.

The most feasible source of memory controller behavior is the analysis of Verilog units used by the compiler to generate the hardware controller, as in this study, but often this approach is rejected because of its tediousness [8]. A more user-friendly source is the use of RTL reports, which shows the type of LSUs to assemble a command request to DRAM [6].

The knowledge of LSUs plus DRAM specifications is combined in this proposal to achieve an accurate memory model that can adjust to changes in memory technology from DDR4 to HBM.

4 HLS Flow for Intel FPGAs

The next subsections present first, the HLS internal details and compilation flow analyzed as part of this work, and second, the performance estimation approach focused on current memory technologies.

4.1 HLS Compilation Flow Internals

Traditionally, hardware description languages (HDLs) were the preferred language for programming FPGA devices, but

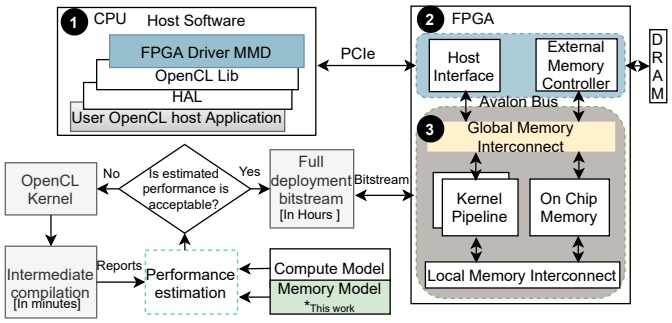


Figure 1. Main elements of OpenCL for FPGAs. ❶ and ❷ represent the BSP, and ❸ the kernel logic.

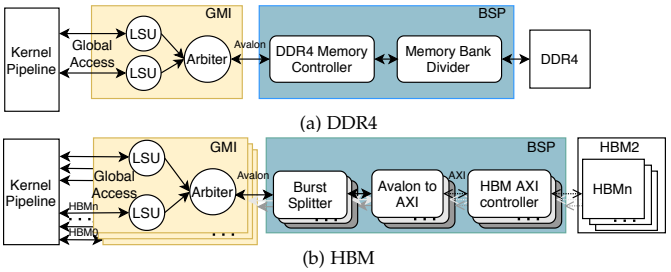


Figure 2. FPGA block units for Intel OpenCL SDK with a) DDR4 and b) HBM memory

the learning curve is steep for the average programmer, hindering wider adoption. Recently, HLS has evolved to a point where programming in languages such as C or OpenCL for FPGA becomes an easier task. The explicit parallelism of the OpenCL programming model offers many opportunities to exploit the pipeline parallelism inherent in streaming processing, making OpenCL a good alternative language for FPGAs.

Figure 1 shows the compilation flow of an OpenCL kernel for FPGAs, which consists of two main steps. First, a translator generates HDL code and RTL reports, called intermediate compilation; and second, a synthesis tool generates the bitstream as flow diagram. Between these two steps programmers can potentially optimize the kernels, based on programming guides and estimation models.

Without loss of generality, the described flow applies to all Intel HLS tools based on the aoc compiler (OpenCL, OneApi, . . .). Figure 1 shows the main components of an OpenCL application. On the host side, ❶, the application communicates with the FPGA device through the board support package [1] (BSP, in blue on the figure). A BSP implements the lower layers of the application stack performing the basic I/O with the board, and the PCI express (PCIe) communications. On the FPGA side, the BSP, ❷, provides support to communicate back with the host, and with the device memory, DRAM and external devices.

The BSP on the FPGA side differs for each FPGA model and each type of external memory, requiring specific intellectual property (IP) controllers and interfaces. For example, for a DDR4 memory with multiple banks, in Figure 2a, the BSP uses the Avalon-MM interface and it has a memory bank divider which can support the interleaving of memory banks for one variable, or uses each bank separately. For HBM memory, Figure 2b, the interface with the BSP is the Advanced extensible Interface (AXI), and the 32 HBM pseudo-channels have a separate GMI and controller because each pseudo-channel works as independent memory using the "heterogeneous memory" feature of the OpenCL compiler although the technology is the same [23].

From a programmer's perspective, the most important component in a BSP is the kernel logic, ❸, which corresponds mainly to the compiled OpenCL kernel. The generated blocks which most critically affect performance are the kernel pipeline and the Global Memory Interconnect, therefore, they are described in detail bellow.

1. Manufacturers often provide BSP, but advanced users can tune and re-implement them.

4.1.1 Kernel Pipeline

The kernel pipeline implements all data and control operations. The high-level OpenCL statements are translated into a graph where each node performs an operation. To receive and send data, there are nodes that interconnect the pipeline with either the local or global memory. To exploit work-item parallelism, HLS tools implement pipelines. Besides pipeline length and initiation interval, splitting up the processing into small pipeline stages also helps to reach higher frequencies, improving kernel and memory performance [24].

4.1.2 Global Memory Interconnect

The GMI manages the kernel-pipeline request to the external memory. In any OpenCL program, each access to a variable in the external memory constitutes a global access. Since global accesses are the main source of kernel stalls, the GMI implements several strategies to maximize external memory throughput and kernel pipeline flow. Architecturally, like other hardware memory interfaces from Intel [25], the GMI has two main components: LSUs, which track in-flight memory operations, and arbiters, which decide on the order of access. Specifically, there are two independent round-robin arbiters one for read and one for write accesses.

Intel FPGA SDK [6], [26] has defined three LSU types for the GMI: burst-coalesced LSU, prefetching, and atomic-pipelined. To understand the access pattern of each LSU, Listing 2 and Table 1 show the code that generates them and their main features; namely, 1) Pipeline, when an LSU can support multiple active requests at a time, 2) Burst, when requests are grouped before being sent to external memory, and 3) Atomic, which serializes the operation and guarantees atomicity, this being omitted from this table to save space because it is only supported by the atomic-pipelined LSU. Note that each one of these LSU features requires greater hardware complexity. Each global access in the source code may translate to one or several LSUs, as Section 5 describes.

```
1  #define N 1024
2  int random_vector[N]={5,1023, 450, 100, ...}
3  __kernel void
4  test_patterns(  global int *restrict x,
5                  global int *restrict z,
6                  constant int *cn )
7  {    int i    = get_global_id(0);
8       int k    = random_vector[i];
9       int out  = 0;  local int lmem[1024];
10      //Code Snippet form Table I
11      z[0]     = out;                    }
```

Listing 2. OpenCL Code for access patterns in Table 1

Table 1. LSU types and their modifiers in global memory interconnect. The code snippets are from Intel FPGA SDK [6].

LSU Type	Description	Pipelined	Burst	Code Snippets[a]
Burst-Coalesced[b]	Requests are grouped into a set of DRAM bursts			
Aligned	Index is contiguous and aligned to page size	✔	✔	`out = x[i];`
Non_Aligned	Index has a modifier not aligned to page size	✔	✔	`out = x[3*i+1];`
Write_ACK	Index to access has dependencies	✔	✔	`out = x[k]; // k is random`
Cache	Index has repetitive dependencies	✔	✔	`for (uint j=0; j<N; j++) z[N*i+j] = x[j];`
Prefetching	Compiled as Aligned Burst-Coalesced	✘	✔	`out = x[i];`
Atomic-Pipelined	Unique LSU for atomic operations	✔	✘	`atomic_add(&x[0], 1);`

[a] Each code snippet corresponds to line 10 in listing 2.
[b] The burst-coalesced type has four modifiers affecting its organization.

Each LSU type provides a different maximum bandwidth, the burst-coalesced LSU with an aligned modifier being the most efficient type on DRAM technology because it maximizes effective bandwidth utilization. Figure 3 shows a read operation generated by a burst-coalesced LSU. Each LSU has a coalescer unit that tries to group continuous memory addresses into a single burst DRAM operation. Next, the read arbiter dispatches this operation to the Avalon Interconnect FIFO in order to issue a DRAM access to the Memory Controller IP through the Avalon Bus. The benefits of bursting come from the DRAM organization [27] because during a read operation at least three commands are required: precharge (PRE), activate (ACT), and read out (RD). PRE opens a row in every bank; ACT then opens a row in a particular bank; and RD reads the burst out back to the controller.

When an LSU receives a requested address, it attempts to group consecutive addresses into a burst, the *burst_cnt* bus size defining the maximum number of burst requests at compilation time, because contiguous access to memory enables the overhead of PRE/ACT commands to be hidden.

In a burst-coalesced LSU, three counters trigger a request to the DRAM: 1) the Burst_cnt bus, that usually corresponds to memory page size, 2) the maximum number of threads allowed to be coalesced, and 3) the time out to minimize stalls in the kernel pipeline when consecutive requests cannot be coalesced. The compiler can modify this LSU depending on the memory access pattern and other attributes [6]; e.g., in the case of data dependencies, the compiler infers a write-acknowledge LSU (ACK) with a work-item level coalescer.

In a *Prefetching* LSU, the behavior is similar to that of a burst-coalesced LSU since it has a continuous access to external memory, but loading data to a register or RAM anticipating a large amount of data. For write operations, it uses a burst-coalesced non-aligned LSU. In high-end FPGAs, such as Stratix 10, the prefetching LSU is not available; then, the compiler generates a burst-coalesced LSU even with exactly the same code as that the Intel SDK provides for the *Prefetching* LSU [2].

The last type of LSU is the *Atomic-pipeline*; Intel provides limited support for 32-bit integers and it does not fully conform with the OpenCL specification version 1.0. *Atomic-pipeline* is considered one of the most expensive functions which might reduce kernel performance and increase the amount of hardware resources, but its usage can simplify a kernel design [28]. In FPGAs with "heterogeneous memories", this LSU is not available.

2. Our assumption is that this behaviour likely depends on the OpenCL SDK version

4.2 Performance Estimation for FPGAs

The kernel pipeline and the external memory accesses directly impact application performance. Kernel pipelines have already been modeled to predict the execution time aiming at the automatization of the compilation process [4], [8], [9]. For pipelines, one key challenge is the selection of the right execution model, choosing between task and ND-Range, because an incorrect choice may increase the execution time by as much as two orders of magnitude [7], [11].

Existing models have simplified the memory component, especially the GMI, losing details that might provide good opportunities for optimization of kernel implementation. Substantial simplification may be valid for old FPGA devices with simple memory organization but does not apply for current models because kernel resources have grown faster than external memory resources; e.g., an Intel Stratix 10 delivers 9 TFLOPS and the newer Intel Agilex delivers 20 TFLOPS, while DRAM has only improved from DDR4 @ 1333 MHz / 2666 Mbps to DDR4 @ 1600 MHz / 3200 Mbps or DDR5 @ 2100 MHz / 4400 Mbps. In terms of performance, these traditional memory technologies are growing slowly compared with FPGA compute resources, which double every generation [29], [30].

Although external memory technologies are evolving, compared with on-chip memory, the throughput of external DRAM banks is still 380 × worse than on-chip, and it is 80 × larger in size [29]. Hence, the prediction of FPGA kernel execution time focuses on kernel pipeline and external memory, ignoring the local memory because, in most situations, its impact is negligible.

A novel memory such as HBM, composed of multi-channel DRAM memory, increases the memory bandwidth and concurrency to maintain sufficient parallelism to support kernel requests. FPGA models such as Stratix 10 MX can reach 450 Gbps with an HBM2 composed of 32 pseudo-channels. The main challenge with HBM for FPGA programmers is application design because the Stratix 10 MX was not designed with a hardware interconnect to enable communication with HBM. That flexibility implies the HLD programmers have to decide how to manage parallel requests in each HBM pseudo-channel [31], [32].

In Intel FPGAs with HLS, as shown in Figure 1, the external memory controller has independent units separate from the kernel logic, where the LSUs have the same behavior on all DRAM models, this making it possible to analyze different memories with the same model.

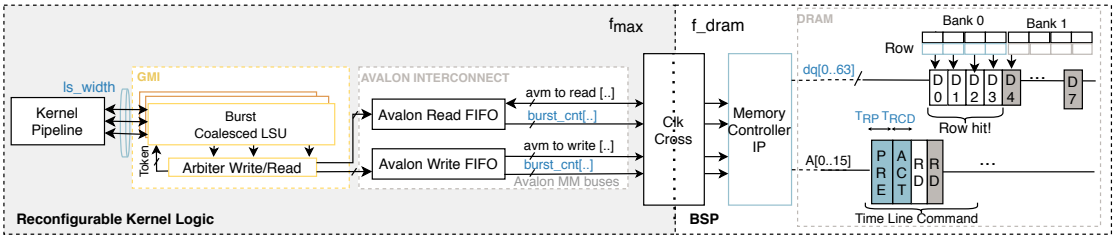


Figure 3. Simplified model of a read operation in a single DRAM bank with an Burst-Coalesced Aligned LSU. The parameter names in blue are used in the model in Table 2

5 Analytical Model

For programs limited by memory, especially bandwidth, the execution time can be estimated accurately by modeling two key components: the GMI, which is the interface between the kernel pipeline and the DRAM memory; and the DRAM memory timing models themselves. The latter has already been modeled by Cho [22], while the modeling of the former, GMI, can be broken down into models of the different LSUs.

Fortunately, the information available after the translation phase, including datasheet and user input, provides enough detail to estimate both GMI and DRAM delays, without the long delays of the full compilation process. During the translation from OpenCL to Verilog, each global access from the kernel source code generates one or several LSUs in the GMI. For each global access, the HLS compiler determines the proper type of LSU according to a static analysis, as described in Section 4.1.

Table 2 summarizes the model input parameters, which are described below:

1) Report: html file which shows the kernel's basic blocks and the LSU types for each global access.
2) Verilog: These files contain the description of the LSU IPs, including key thresholds such as max_th^i.
3) Code: High level source code provides static information about the access and iteration space for loops in order to estimate the number of memory accesses.
4) Datasheets: The DRAM datasheets provide the timing and the organization of DRAM memory chips.

The report and Verilog source files are available after the intermediate compilation stages using *aocl -rtl*. Note that all variables of each source are static and can be automatically recovered. Except the ls_acc which depends on loops, whose iteration space could dynamically vary depending on an input parameter. In such a case, the compiler could not automatically retrieve ls_acc value, and the model should rely on user hints.

To begin with, let T_{est} be the estimated execution time of memory intensive applications. With multiple DRAM banks accessed in parallel, the slowest bank time access T_{bank_n} determines the total execution time, such that:

$$T_{est} = \max_{n=1,\dots,\#banks} T_{bank_n} \tag{1}$$

where T_{bank_n} represents the total delay of the *n-th* DRAM bank estimated as the sum of the minimum time, T^i_{ideal}, plus the overhead time, T^i_{ovh}, from every transaction from every LSU, as shown in (2). While T^i_{ideal} only depends on maximum memory data transfer capacity and hence is the same for all LSU types, T^i_{ovh} varies with the type of LSU, as the next subsections describe.

Table 2. Description of model parameters. The *param* label for the Verilog source refers to a variable name in a Verilog instance

Source	Variable	Definition
Report	$\#lsu$	Number of load-store units units per bank
	ls_width^i	Memory width of i LSU [bytes]
	f_{max}	Estimated kernel frequency [Hz]
Verilog	$burst\_cnt^i$	Size of Avalon $burst\_count$ port *param*:BURSTCOUNT_WIDTH
	max_th^i	Maximum threads in a burst *param*:MAX_THREADS
Code	δ	Address stride of memory access
	ls_acc^i	Number of access of i LSU
	ls_bytes^i	Bytes of a single ls_acc
	v	Kernel vectorization
Datasheet	$\#banks$	Number of banks in parallel
	dq	Memory data width [bytes]
	bl	Memory burst length
	f_dram	Memory frequency [Hz]
	T_{RCD}	Row activation time [s]
	T_{RP}	Precharge row miss time [s]
	T_{WR}	Time to recovery from Write [s]

$$T_{bank_n} = \sum_{i=1}^{\#lsu} \delta^i \cdot (T^i_{ideal} + T^i_{ovh}) \tag{2}$$

where the δ^i factor represents the stride of an access. Regardless of the stride, LSUs always request to DRAM a whole burst of consecutive data, and upon reception, the LSUs discard part of the data burst, increasing the number of memory transactions; e.g., a stride of two discards half of each data burst and doubles the number of accesses.

Assuming a minimum time for fetching all data for the i LSU, T^i_{ideal}, this time can be estimated as the size in bytes, ls_bytes^i multiplied by the number of accesses, ls_acc^i, divided by the kernel memory bandwidth, bw_mem^i, as shown in (3).

$$T^i_{ideal} = \frac{ls\_bytes^i \cdot ls\_acc^i}{bw\_mem^i} \tag{3}$$

All these equations are valid for memory-intensive applications that can saturate the available memory bandwidth. At this point, the addition of more compute resources does not provide any benefit because execution time is already

dominated by the DRAM bank access delay. When the kernel-pipeline clock frequency, f_{max}, is higher than the required minimum frequency for each LSU, f^i_{min}, then the memory bandwidth is saturated. In that case, the bw_mem^i reaches the maximum bandwidth. Otherwise, the memory bandwidth is non-saturated bw^i_{nsat} as (4) shows.

$$bw\_mem^i = \begin{cases} bw\_dram & f_{max} \geq f^i_{min} \\ bw^i_{nsat} & otherwise \end{cases} \quad (4)$$

To satisfy the memory bandwidth saturation condition, the kernel needs a minimum DRAM memory data request size of ls_width^i, for each i LSU, noting that ls_width^i cannot be greater than DRAM burst $dq \cdot bl$. The memory bandwidth saturation for double data rate DRAM is $bw\_dram = dq \cdot 2 \cdot f\_dram$, where f_dram is the DRAM frequency. The ratio of bw_dram to ls_width^i describes the relation between kernel-pipeline requests and external memory capacity, defined as f^i_{min}, in (5).

$$f^i_{min} = \frac{bw\_dram}{ls\_width^i} \cdot \delta^i \quad (5)$$

The modifier δ^i increases the memory burst requests, and therefore, the kernel-pipeline requirements for memory bandwidth.

Although f_{max} is estimated in the intermediate compilation, it could be inaccurate as the wire delay is not considered [2]. The increase in kernel-pipeline resource usage and algorithm complexity could reduce the reported f_{max} after synthesis.

When f^i_{min} is less than f_{max}, the memory bandwidth is non-saturated, and two cases are possible: first, the number of LSUs, $\#lsu$, per memory bank is equal to one, and second, $\#lsu$ is greater than one, in this case, the kernel fully exploiting the double rate memory frequency, multiplying the f_max by two. Finally, bw^i_{nsat} is a portion of the relation between f_{max} and f^i_{min} defined in (6).

$$bw\_nsat^i = \begin{cases} bw\_dram \cdot \frac{f_{max}}{f^i_{min}} & \#lsu = 1 \\ bw\_dram \cdot 2\frac{f_{max}}{f^i_{min}} & \#lsu > 1 \end{cases} \quad (6)$$

The relation between f_{max} and f^i_{min} shows the "clock crosser" influence on two different clock frequency domains between kernel-pipeline and memory controller, as shown in gray and white boxes on Figure 3. This is evidence that the effective DRAM memory bandwidth in a FPGA could be modified after the compilation process if the f^i_{min} condition is not satisfied.

While this work is focused on bandwidth saturated programs, the non-saturated memory bandwidth includes compute cycles, and these have already been covered [8], [9]. Given bw_mem, the model can predict whether this new model should be used to estimate the execution time or previous compute-oriented models would be preferable, as set out in (7).

$$\text{Kernel Bound} \Rightarrow \begin{cases} \text{Memory saturated} & bw\_mem^i = bw\_dram \\ \text{Memory non-saturated} & \text{otherwise} \end{cases} \quad (7)$$

Finally, once a kernel is defined as memory saturated, T_{est} can be calculated with (1).

5.1 Burst-Coalesced LSU

The burst-coalesced LSU is one of the main types of GMI, as listed in Table 1 in Section 4.1. In this LSU type, in order to saturate memory bandwidth, the Avalon FIFO needs to be filled with requests. When the kernel pipeline does not make enough requests to fill the memory burst before time out, the memory bandwidth is non-saturated.

It is possible to achieve T_{ideal} for contiguous memory accesses, this type of access hiding PRE/ACT latencies, as was shown in Figure 3. Furthermore, bank-interleaving memory controllers can completely hide the opening of new memory banks [33] while the $\#lsu$ remains below two. When the $\#lsu$ increases, this forces the DRAM to open a new row, adding T_{ovh}.

The T^i_{ovh} is proportional to DRAM latency of opening a new page, given by row miss commands (T_{row}). These can be calculated based on the number of times that an i LSU has to open a new row, which depends of the number of burst transactions, with a given $burst\_size$, required to request the total number of bytes ($ls\_acc \cdot ls\_bytes$), formulated as in Equation (8). It should be noted that LSU latency and the amount of data in the Avalon FIFO would hide the kernel latency, and for this reason, only the DRAM latency is considered.

$$T^i_{ovh} = \begin{cases} 0 & \#lsu \leq 2 \\ \frac{ls\_acc^i \cdot ls\_bytes^i}{burst\_size^i} \cdot T_{row} & \text{otherwise} \end{cases} \quad (8)$$

The estimation of $burst\_size$ and T_{row} for each LSU modifier are analyzed in Subsections 5.1.1 to 5.1.3.

5.1.1 Burst-Coalesced Aligned LSU

This modifier is generated when all the kernel requests are memory addresses aligned to page size, buffering contiguous memory requests until the largest possible burst, or DRAM page size, can be made [26]. Where multiple load/store requests are consecutive words to memory, the burst-coalesced aligned LSU maximizes the memory throughput. The complete architecture of this LSU for a load and store request is shown in Figure 3

Here, to estimate T_{row}, the DRAM $burst\_size$ is defined as the size of burst transaction, which can overlap DRAM commands. DRAM sets the minimum burst transaction size to $dq \cdot bl$, but it can transfer multiple consecutive burst for the same open row yielding (9), where $burst\_cnt^i$ represents the bus size of the transaction counter, as shown in Figure 3.

$$burst\_size^i = 2^{burst\_cnt^i} \cdot dq \cdot bl \quad (9)$$

The estimation of T_{row} is not trivial because the controller can overlap commands due to reordering strategies and the page policy [34]. This model takes into account the inter-command delay for row buffer misses [9] using ACT/PRE latencies, as (10) shows. The command sequence PRE and ACT, for read and write, is considered with the same minimum timing as the FPGA profile shows a minimal bandwidth difference between operations.

$$T_{row} = T_{RCD} + T_{RP} \tag{10}$$

In a kernel, each global access variable reduces the memory bandwidth with increases in T_{ovh}. The overhead is only zero if the accesses to DRAM banks are consecutive. But if global accesses are to different addresses, as in the case of a multiple global access pointer, the accesses are not consecutive and therefore T_{ovh} appears. Based on this model, we can make a first observation:

> **Observation 1:** Each variable in the same DRAM bank adds an overhead of T_{row}. This time is null using one bank per global access; for example, with multiple DDR4 banks, manually distributing the data buffers and disabling the interleaving with the compilation flag *-no-interleaving*; and with HBM using one variable per pseudo-channel.

5.1.2 Burst-Coalesced Non-Aligned LSU

Both aligned and non-aligned LSUs try to coalesce requests from multiple threads in a single burst command; however, the δ stride of non-aligned access adds a new trigger for a memory request, the number of threads, max_th, that have been launched and coalesced in one memory request.

Equation (11) calculates this constraint, called max_reqs, representing the maximum size of a DRAM request. When a coalescer assembles a request, either the request occurs when the amount of data requested is equal to a DRAM page or when the number of coalesced requests have reached max_th, defined as a constant in the LSU Verilog source code. This limit is affected by δ, it reducing the effective burst request. In the other case, the δ fraction of ls_width is the effective burst size, as (12) shows. Note that ls_width should be bounded by DRAM page size.

$$max\_reqs^i = \frac{max\_th \cdot ls\_width^i}{\delta + 1} \tag{11}$$

$$burst\_size^i = \begin{cases} \frac{max\_reqs^i}{\delta} & max\_reqs^i \leq 2^{burst\_cnt^i} \cdot dq \cdot bl \\ \frac{ls\_width^i}{\delta} & \text{otherwise} \end{cases} \tag{12}$$

Based on this model, we can make a second observation:

> **Observation 2:** The stride value δ multiplies the number of memory accesses required, it being able to saturate DRAM bandwidth with discarded data.

5.1.3 Burst-Coalesced Write-Acknowledge LSU

When the global access includes data dependencies in its indexation, the compiler generates a write acknowledgement signal to guarantee the correct ordering of accesses [6]. Therefore, the burst size equals the aligned case from Equation (9), and most important, each burst only consumes ls_bytes increasing the total time by $\frac{dq \cdot bl}{ls\_bytes}$. The write-ack signal adds a write command to the DRAM access, increasing the T_{row} delay as (13) shows. Based on the write-acknowledge LSU model, we can make a third observation:

$$T_{row} = T_{RCD} + T_{RP} + T_{WR} \tag{13}$$

> **Observation 3:** Although the compiler detects access dependencies, the logic tries to generate a burst request to DRAM, because each FPGA cycle at a minimum frequency of $\frac{2 \times F_{mem}}{bq}$ is equivalent to 1 burst request to DRAM.

5.2 Atomic-pipelined LSU

The atomic-pipelined LSU executes a read and a write DRAM command. It only supports integer data types without bursting (therefore, in (2), $\delta = 1$). For example, `atomic_add` from Listing 3 atomically sums `val` to `p`, which is atomically read and written. When `val` is constant within a loop or for multiple work items, then the compiler performs v operations atomically.

Atomic operations cannot be used with multiple memory interfaces as is the case of HBM2 because BSP does not provide support for them.

```
1 int atomic_add(volatile __global int *p, int val);
```

Listing 3. Atomic-pipelined add prototype function

Equation (14) shows the resulting T^i_{row}, including the two accesses, and T^i_{ovh}, depending on the vectorization factor v. Note that memory saturation in atomic should include the LSU as a unique operation (sum of ls_width) of two LSUs. Based on atomic, we can make a fourth observation:

$$\begin{aligned} T^i_{row} &= 2 \cdot (T_{RCD} + T_{RP}) + T_{WR} \\ T^i_{ovh} &= \begin{cases} \frac{T^i_{row}}{v} & \text{val is constant} \\ T^i_{row} & \text{otherwise.} \end{cases} \end{aligned} \tag{14}$$

> **Observation 4:** The atomic LSU is the most time expensive LSU because one FPGA cycle performs only one atomic operation. Atomics are limited to *int* data types.

6 Methodology

The experiments have been run on two FPGAs with different memory technologies: an Intel Stratix 10 GX Development Kit with 2 GB of DDR4 DRAM HiLo running at 1866 MHz [35] and an Intel Stratix 10 MX Development kit with a HBM2 memory with 32 pseudo-channels, each one with 256 MB of capacity running at 800 MHz [31], [36]. Table 3 shows the parameters required for the model on each FPGA. The other parameters come from the intermediate compilation of the Intel FPGA SDK for OpenCL 18.1 for Stratix10 GX and 19.3 for Stratix 10 MX. The OpenCL versions are different because the manufacturers designed the BSPs with different Quartus IP versions.

To validate the model, two types of benchmarks are analyzed: first, a set of microbenchmarks, targeting each LSU type from Table 1 inside Listing 4, where user parameters such as v and the number of global access ($\#ga$) vary. For DDR4 memory on Stratix 10 GX, only one bank is available, and in the Stratix 10 MX with HBM memory, the "heterogeneous memory" feature is used, this assigning each global access to an HBM bank. For burst-coalesced aligned and non-aligned LSUs, δ variations are validated scaling

Table 3. Fixed variable value to evaluate the LSU model on Stratix 10 GX and Stratix 10 MX with a DDR4 1866 and HBM2 memory respectively. All variables (Var.) are defined in Table 2

Memory	Var.	Value	Var.	Value
DDR4-1866 [35]	f_dram	933.3 MHz	T_{RCD}	13.5 ns
	dq	8 B	T_{RP}	13.5 ns
	bl	8	T_{WR}	15.0 ns
HBM2 [31], [36]	f_dram	800.0 MHz	T_{RCD}	14.0 ns
	dq	8 B	T_{RP}	14.0 ns
	bl	4	T_{WR}	15.0 ns

the array accesses by δ. In the non-aligned case, an offset argument is added to the scaled index forcing the compiler to this LSU.

```
#ifdef HBM // for HBM with multiple banks
  #define g_bank(global_mem_label) \
  __attribute((buffer_location(global_mem_label)))
#else // for DDR4 memory
  #define g_bank(global_mem_label)
#endif
__attribute((num_simd_work_items(SIMD)))
__kernel void test_coalesced(
  __global g_bank(HBM1) const int *restrict x0,
  ..
  __global g_bank(HBMn) const int *restrict xn,
  __global g_bank(HBM0) const int *restrict z)
{
    int id = get_global_id(0);
#ifdef Burst_Coalesced_Aligned
    z[id] = x1[id] + ... + xn[id];
#elif Burst_Coalesced_Non-Aligned
    z[3*id+1] = x1[3*id+1] + ... + xn[3*id+1];
#elif Burst_Coalesced_Write-Acknowledge
    int idr = rand[i]; //work item index
    z[idr] = x1[idr] + ... + xn[idr];
#elif Atomic
    atomic_add(&z[0], x[id]);
    ...
    atomic_add(&z[n], xn[id]);
#endif
}
```

Listing 4. OpenCL template microbenchmark to vary global access number

A second validation is performed with 18 different HPC benchmarks, all memory bound, selected from the following sources: Intel FPGA SDK, Xilinx SDAccel, NVIDIA OpenCL, Rodinia FPGA [7], Chai [10], and FBLAS [37], in which input channels were modified to fit the DRAM inputs.

The execution time is measured with *aocl -report* enabled with profiler compilation, this setting up the hardware counter in the LSU. The atomic cases are measured with OpenCL events since this type of LSU does not have dynamic counters implemented.

7 Model Validation

The model validation comprises two sets of experiments. The first, microbenchmarks, includes small programs with multiple configurations of kernel v, δ, and $\#lsu$, enabling us to understand how each parameters affects performance in isolation. The second set is made of complete benchmarks to test the model with well-known applications. A third set of experiments are conducted to compare our proposals with previous ones [8], [9].

The model assumes that in memory saturated applications, the execution time depends more on memory delay than on kernel frequency; this is valid provided that the kernel frequency is high enough for the memory controller to fully exploit bandwidth, namely, f_{min}. To verify this claim, Figure 4 shows the execution time for multiple vector addition kernels with burst-coalesced aligned LSU (line 16 of Listing 4) varying $\#lsu$ and v in DDR4 and HBM2 memories[3]. For memory saturated kernels (points circled in red), f_{max} does not affect execution time regardless of $\#lsu$ and v, as the flat dashed curves clearly indicate, because the memory delay dominates execution time as Equations (4) to (7) show; e.g., in DDR4 with $\#lsu > 3$, the flat trend reflects only minor variations in execution time. In HBM, the trend is less visible because the axis values are overlapped in cases of bandwidth saturation, indicating the independence of time from v. For non-saturated memory bandwidth kernels, points not circled, ls_width^i, set by v, affects performance more than f_{max} since these cases do not satisfy the f_{min} condition. Both results point out that programmers are able to estimate how well the memory bandwidth is exploited based on ls_width and f_{max} from reports since the results show the dependency on these parameters.

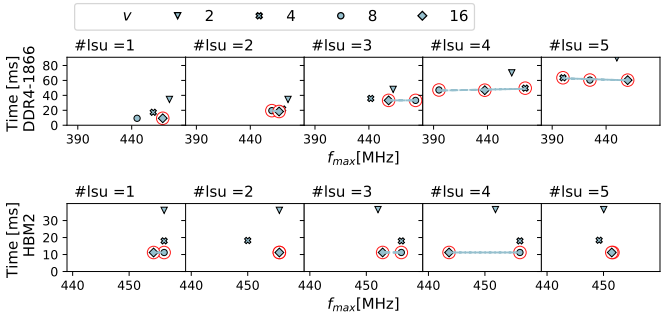


Figure 4. Execution time vs. kernel frequency f_{max} after synthesis with a burst-coalesced aligned LSU varying $\#lsu$ and the vectorization factor (v) in DDR4 1866 and HBM2 memories. The points circled in red correspond to saturated memory kernels and the dashed lines indicate the flat trend of memory saturated kernels.

7.1 Microbenchmarks

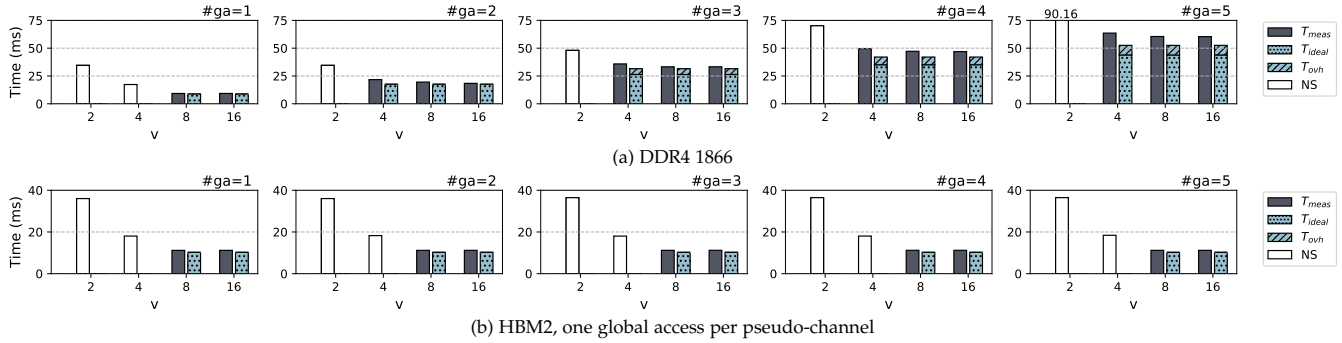
For the sake of completeness, each LSU modifier is evaluated separately. The evaluation comprises the microbenchmark from Listing 4 with their body tuned to the LSU type and modifier. Every loop body is based on vector addition to easily change $\#ga$.

Note that in HBM2 memory each global access has a single pseudo-channel to parallelize bank access, while in DDR4, multiple global accesses must be arbitrated by a controller.

7.1.1 Burst-Coalesced Aligned LSU

Investigating each LSU type in more detail, Figure 5 compares the measured, T_{meas}, and analytically estimated, T_{est}, execution times for a burst-coalesced aligned LSU. For T_{est}, each bar corresponds to the sum of T_{ideal} (dotted) and T_{ovh} (striped). For HBM2, the slowest bank from Equation (1) is shown, while DDR4-1866 has only one bank. With this LSU type, each global access generates one LSU ($\#ga$ is equal to $\#lsu$).

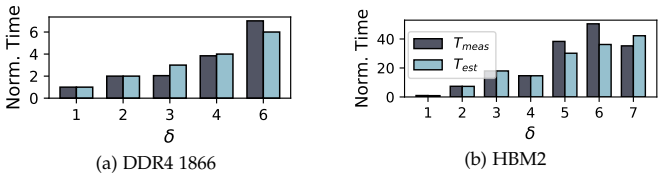
3. The other LSU types produce the same results and these are not shown for clarity and brevity.



(a) DDR4 1866

(b) HBM2, one global access per pseudo-channel

Figure 5. Measured (T_{meas}) and estimated ($T_{ideal} + T_{ovh}$) time for the burst-coalesced aligned LSU varying the vectorization factor (v) and global access ($\#ga$) in two types of external memory: a) DDR4 1866 and b) HBM2. The bars with dots and stripes represent T_{ideal} and T_{ovh}, respectively. Kernels with non-saturated memory bandwidth (NS) are detected (empty bars) and not estimated.



(a) DDR4 1866

(b) HBM2

Figure 6. Measured (T_{meas}) and estimated (T_{est}) time are normalized to T_{meas} for $\delta = 1$. The experiment varies δ with $\#lsu = 3$ and $v = 16$ for burst-coalesced aligned LSUs in two types of external memory: a) DDR4 1866 and b) HBM2, adjusting for special cases.

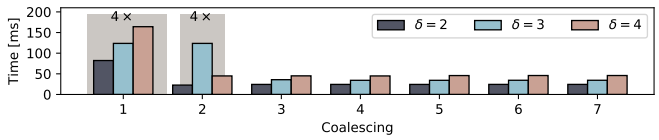


Figure 7. Measured time (T_{meas}) varying δ values in the burst-coalesced aligned LSU increasing the number of coalesced accesses. The gray shading marks the special cases where the model needs to be adjusted for HBM2 memory.

For all cases, errors remain below 15%, the simplification of the DRAM commands in the model and the refresh time being among the main sources of error, which can reduce memory efficiency, e.g., the DDR4 IP controller reduces efficiency by around 3.5% [33]. The experiment also evidences that the higher the $\#lsu$, the higher the T_{ovh}; e.g., DDR4 bandwidth reduces by 26%, from 14.2 to 10.5 GB/s with five LSUs. Hence, in this case, *Struct of Array* is a good option for reducing $\#lsu$. In the case of HBM2 memory, the time remains the same with the increase in $\#lsu$ because they run in parallel with one LSU per pseudo-channel.

Figure 6 shows the times, normalized to T_{meas} with $\delta = 1$, for multiple stride values. Execution time shows a linear dependency on δ because of the data discarded in each DRAM burst.

Notice that burst-coalesced aligned LSU cannot be generated with all δ values because the compiler does not detect DRAM page alignment. With HBM2 memory, strided write operations need a correction factor of 4 because they do not detect coalescing, and the burst splitter divides the request into $bl = 4$ words inside a burst taking bl cycles to transfer it, while a read request only needs one clock cycle for bl words. The write stride HBM2 exception in Figure 7 shows a sweep of δ values varying coalescing, which is added with more contiguous memory access in the main loop in Listing 4. The shading indicates the special cases where the execution time is 4 × longer. Comparing stride access in DDR4 and HBM2, the performance of HBM2 is lower, by 2 × in the worst case, starting from $\delta = 2$ due to bursts splitting in store for HBM, in spite of parallels between the three LSUs used in this test.

7.1.2 Burst-Coalesced Non-Aligned LSU

The burst-coalesced non-aligned LSU is depicted in line 17 of Listing 4 for a $\delta = 3$. Similar to the aligned modifier, in this case, the global access is also supported by just one LSU. burst-coalesced non-aligned LSU, in Figure 8, shows a 22% larger error than burst-coalesced aligned LSU, this being attributable to the latency of the coalescer having a large variance; e.g., the number of required address comparisons depends on the coalescer state. The largest errors, as with burst-coalesced aligned LSU, are related to small vectorization factors; in the case of DDR4 with v=4, the calculated f_{min}=349 MHz compared with f_{max} after compilation which is in the range of 301 to 418 MHz placing the kernel near to a non-saturated memory state and increasing the minimum error by 13%. Also note that neither v nor $\#ga$ correlates with the error.

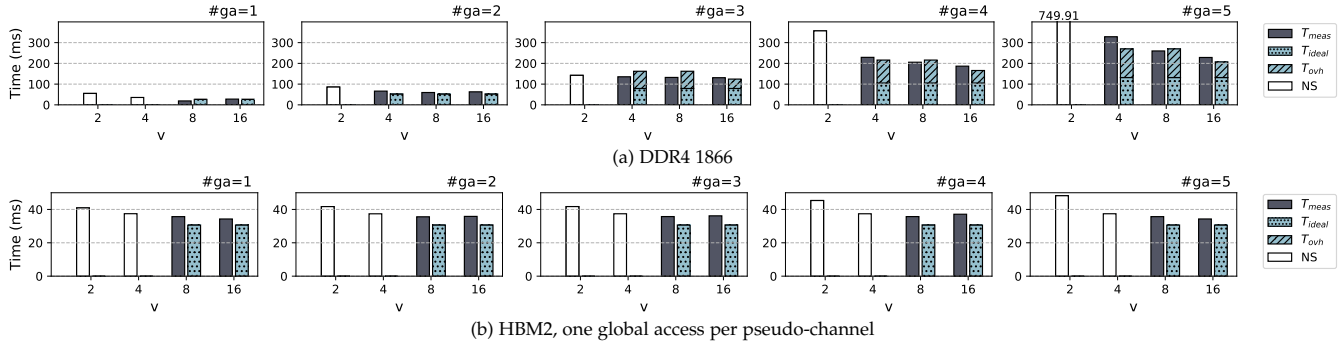
Further, for v and $\#ga$ larger than 4 and 3, respectively, the number of threads in a burst, max_th of Equation (12), significantly impacts execution time, which increases linearly and not exponentially like v. This "max_th effect" can also be seen varying δ as Figure 9 shows for $v = 16$ and $\#lsu = 3$, with times normalized to $\delta = 1$. For $\delta = 7$, the max_th restriction appears optimizing the access that increases with strides. Compared to an aligned LSU, the performance is 60% lower on average due to address comparison increases and the burst window being reduced to avoid long kernel stalls.

Unlike in DDR4, in HBM2, the execution time does not have T_{ovh}, as in the burst-coalesced aligned LSU case, due to the use of just one LSU per pseudo-channel. It should be noted that $\#ga$ does not vary the estimation results, showing independence between HBM channels.

7.1.3 Burst-Coalesced Write-Acknowledge LSU

The evaluation of this LSU type uses the microbenchmark in Listing 4, with the code snippet from lines 20 to 21 of.

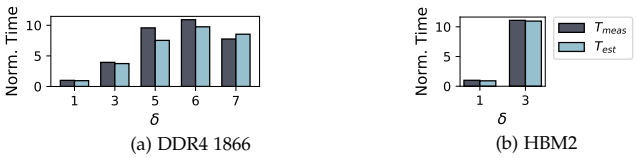
An array of constant values is generated by software with random values between 0 and 2048, reducing the probabil-



(a) DDR4 1866

(b) HBM2, one global access per pseudo-channel

Figure 8. Measured (T_{Meas}) and Estimated ($T_{ideal} + T_{ovh}$) time for the burst-coalesced non-aligned LSU varying the vectorization factor v and global access ($\#ga$) in two types of external memory: a)DDR4 1866 and b) HBM2. Kernels with non-saturated memory bandwidth (NS) are detected (empty bars) and not estimated.



(a) DDR4 1866 (b) HBM2

Figure 9. Measured (T_{meas}) and estimated (T_{est}) time are normalized to T_{Meas} in $\delta = 1$. The experiment varies δ with fixed values of $\#lsu = 3$ and $v = 16$ for burst-coalesced non-aligned LSU in two types of external memory: a) DDR4 1866 and b) HBM2.

ities of coalescing (2048 over 64 and 32 floats coalesced in DDR4 and HBM2 respectively).

In the previously analyzed LSU types (burst-coalesced aligned LSU and burst-coalesced non-aligned LSU), v affected the lsu_width; by contrast, with write-acknowledge LSU, the lsu_width remains constant. To increase the vectorization, the compiler generates as many LSUs as the desired v for each global access. The assumption is that every thread is accessing a different memory location, controlling the memory consistency with the ACK signal. Figure 10 shows the comparison between the measured and estimated execution times.

Among all burst-coalesced LSU modifiers, write-acknowledge LSU is the one that penalizes performance the most, growing $24\times$ more than with burst-coalesced aligned LSU. The read operations show a stall on read until 98% with two LSUs. To optimize these cases, the programmer should evaluate a balance between the data dependency with writes vs. the use of on-chip memory with a tiling strategy.

For HBM2, the assumption of this LSU type is replaced by a burst-coalesced aligned LSU tree, but as write-acknowledge LSU uses a signal to control pipeline flow. Here, the estimation has a maximum error of 12% for all $v = 2$ values, as with burst-coalesced non-aligned LSU, kernel-pipeline is near to memory saturation with a $f_{min} = 400$ MHz and a minimum kernel frequency after compilation of 367 MHz, compared with expected value of 450 MHz.

7.1.4 Atomic-pipelined LSU

The evaluation of this LSU type uses the microbenchmark in line 23 of Listing 4. In this code, to generate a single global access ($\#ga = 1$), the global access $xn[id]$ is replaced by a local variable id. Otherwise, each atomic operation generates one global access per v to avoid coalescing. Only DRAM results are shown, because the atomic LSU is not supported with HBM2 memory.

In general, the atomic-pipelined LSU does not change the lsu_width, unlike the burst-coalesced LSU, making T_{ovh} the most significant component in the case of this LSU. Figure 11 shows that execution time increases linearly with $\#ga$, the maximum error of 16% corresponds to unaccounted 5 ns per atomic operation. The hypothesis is that this delay is close to the time between the beginning of the internal write transaction and that of the following read command in the same group and same bank (T_{WTR}).

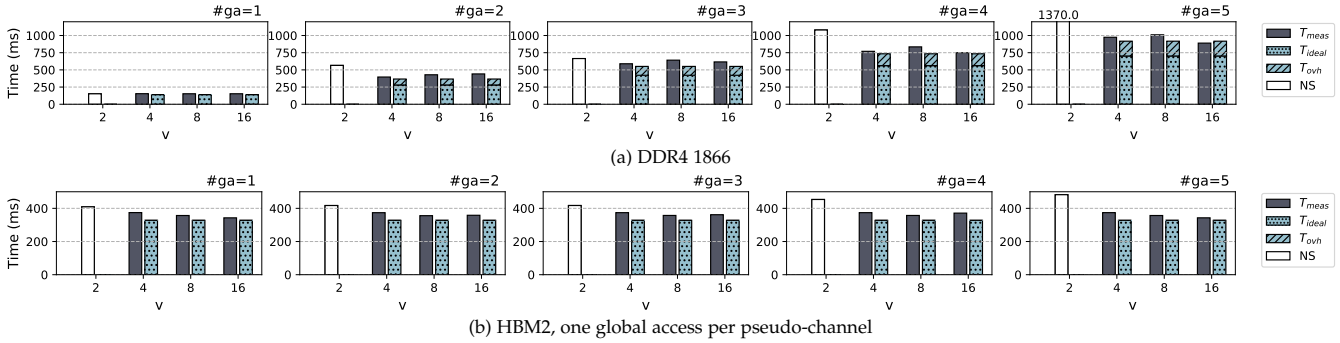
Overall, analyzing read stalls quantifies the impact of the LSU on kernel performance. For burst-coalesced aligned and non-aligned LSUs, the read stall percentages are under 20% because the coalescer partially hides the δ-induced delay. Meanwhile, write-acknowledge LSU has a stall percentage of over 50% as the extra signalling serializes the requests. The atomic-pipelined modifier cannot be measured because profiling is unsupported, but it is safe to assume that stalls will be high due to atomicity requirements.

7.2 Applications

To cover a large set of possible scenarios, this section evaluates the model with 18 bandwidth bound applications, mixing single task and NDRange kernels with and without channels. Table 4 reports the measured and estimated times with the corresponding errors for all of them.

For all the applications with a DDR4-1866, the relative error remains below 9.2% with an average value of 7.6%. With HBM2 memory, the error is higher, with a maximum of 55%.

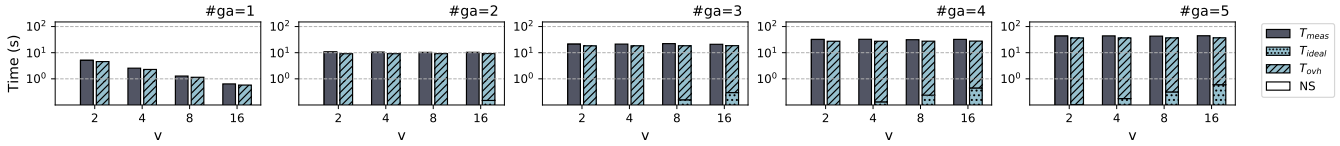
The main source of error in HBM is the frequency requirements from the controller, which needs $f_{min} = 400$ MHz to maximize bandwidth. Such an f_{min} is difficult to achieve with high resource usage that increases the pressure on the place-and-route compilation phase and reduces the achievable target frequency f_{max} [24]. For example, in MatrixMult with $v = 128$, the kernel requires the highest (53%) DSP resource allocation among benchmarks, the compilation time is around 9 h, and the kernel only achieves a $f_{max} = 177$ MHz, 55% lower than the expected f_{min}. Further, MatrixMult with $v = 64$ uses 26% of DSP blocks, takes 5 h to compile, and yields an $f_{max} = 268$ MHz, 33% lower than f_{min}. If future HLS tools improved place-and-route capabilities, errors would certainly decrease.



(a) DDR4 1866

(b) HBM2, one global access per pseudo-channel

Figure 10. Measured (T_{meas}) and estimated ($T_{ideal} + T_{ovh}$) time for burst-coalesced write-acknowledge LSU varying the vectorization factor v and global access ($\#ga$) in two types of external memory: a) DDR4 1866 and b) HBM2. Kernels with non-saturated memory bandwidth (NS) are detected (empty bars) and not estimated.



(a) DDR4 1866

Figure 11. Measured (T_{meas}) and estimated ($T_{ideal} + T_{ovh}$) time for Atomic-pipelined LSU varying the vectorization factor (v) and global access ($\#ga$) in a DDR4 1866 memory. Non-saturated memory bandwidth (NS) are detected (empty bars) and not estimated. The time axis is in seconds and logarithmic.

Table 4. Kernel applications and estimated time in two memories: DDR4 1866 and HBM2.GMI- global memory interconnect BCA- burst-coalesced aligned LSU. BCNA- burst-coalesced non-aligned LSU. ACK- burst-coalesced write-acknowledge LSU. M- Measured. E- Estimated

			DDR4 1866				HBM2			
Kernel	GMI	$\#lsu$	BW [GB/s]	M.Time [ms]	E.Time [ms]	Error [%].	BW [GB/s]	M.Time [ms]	E.Time [ms]	Error [%]
axpy [37]	BCA	3	11.8	31.9	31.5	1.2	34.5	11.2	10.2	8.6
Dot [38]	BCA	3	13.3	29.4	31.5	7.3	23.4	11.2	10.2	8.5
FFT-1D Direct [6]	BCA	2	13.8	9.5	8.8	7.3	19.8	6.6	5.1	22.4
FFT-1D Inverse [6]	BCA	2	13.8	9.5	8.8	7.4	19.6	6.6	5.1	23.4
iamax [37]	BCA	2	14.3	9.2	8.8	4.6	11.7	11.2	10.2	8.6
nn [7]	BCA	2	13.9	11.0	10.3	6.5	17.6	8.7	8.0	8.8
PrefixSum [11]	BCA	2	12.7	10.0	9.0	10.1	23.1	5.2	5.8	9.6
ROT [37]	BCA	4	11.6	35.7	39.5	10.6	47.9	11.5	10.5	8.7
Sobel Filter HD [39]	BCA	3	13.2	1.9	2.0	6.2	14.1	1.7	CB	-
VectorAdd [6]	BCA	3	12.1	33.3	33.2	5.1	35.9	11.2	10.2	8.6
VectorAdd$\delta = 2$	BCA	3	5.9	67.9	63.0	6.5	4.7	82.6	81.9	0.8
Histogram [10]	BCA	2	14.3	8.9	8.4	5.8	9.2	13.4	9.8	26.6
Hotspot [7]	BCNA	3	7.5	9.7	8.8	8.7	16.9	12.8	CB	-
MatrixMult (v=64) [6]	BCNA	3	8.9	31.2	27.9	10.3	17.6	15.7	10.4	33.3
MatrixMult (v=128) [6]	BCNA	3	9.1	121.2	107.8	11.0	11.6	94.4	41.9	55.6
Pathfinder [7]	BCNA	3	7.6	27.6	25.4	7.9	11.6	13.8	16.5	20.2
WM [40]	BCNA	2	13.9	59.8	55.8	6.6	12.6	0.2	0.2	6.2
NW [7]	BCNA/ACK	4	0.3	1.4	1.4	4.0	0.3	0.2	0.2	25.7

To illustrate the frequency differences in DDR4 and HBM2 between applications, the histogram in Figure 12 shows the applications distribution in terms of frequency and marks the minimum frequency required to maximize memory bandwidth. On HBM2, 8 applications are critically bounded by the frequency after place-and-route because they do not reach the pre-synthesis reported f_{max}. Figure 13 analyzes these 8 applications and shows the post-synthesis time and frequency error compared to the estimated pre-synthesis values. There is a strong correlation between time and frequency error, suggesting that compiler accuracy estimating the frequency can limits the model's accuracy. As a especial case, the Stratix 10 MX with HBM2 memory requires higher frequency to saturate memory. In this device, the frequency estimation worsens compared to that of the GX because the MX BSP uses 32 separate global memory interfaces connecting to the physical pseudo-channels with 256-bits buses. In fact, the worst estimation time and the worst frequency estimation from the tool comes from MatrixMult in where the routing tool reports routing congestion warning.

7.3 Comparison with other models

This subsection compares the proposed model with two state-of-the-art models: Wang and HLScope+ [8], [9], reproducing the mathematical models for the microbenchmarks, with $f = 16$, and for the vectorAdd application. Unfortunately, comparison with other applications is unfeasible because the dynamic profiling tools feeding Wang and HLScope+ are not available. The tests are run with two BSPs for Stratix 10 GX with different DRAM frequencies, 1866 and 2666 MHz.

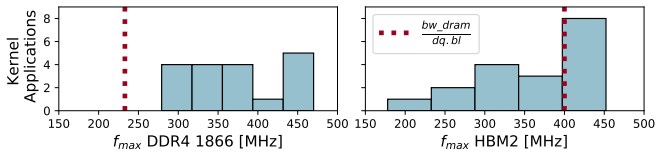


Figure 12. Frequencies Histogram for 18 kernel applications; the red dotted line shows the required minimum frequency for maximizing bw_dram.

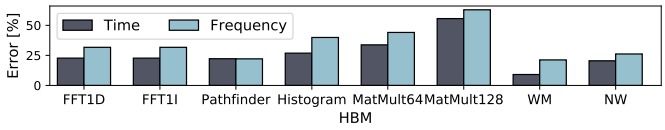


Figure 13. Estimation error of the execution time and frequency error from pre-synthesis report and after place-and-route in kernels that are limited by a frequency under $400MHz$ after synthesis in HBM cases.

In all but one case, μb burst-coalesced aligned LSU, the error found in this study is lower than that of Wang and HLScope+ as Table 5 shows. Comparing the maximum error of each model, this proposal is up to 400 and 5 $\times$ more accurate than Wang and HLScope+, respectively.

Table 5. Execution time estimated error; μb, BCA, BCNA, and ACK refer to microbenchmark, burst-coalesced aligned, burst-coalesced non-aligned, and burst-coalesced write-acknowledge LSUs, respectively.

Benchmark	$\#lsu$	Wang [%]	HLScope+ [%]	This work [%]
DDR4-1866				
μb BCA	1	17.3	12.7	5.6
μb BCA	4	0.3	10.6	4.4
μb BCNA	3	-	71.1	4.0
μb ACK	32	8049.9	63.2	27.9
VectorAdd	3	19.3	21.0	5.1
DDR4-2666				
μb BCA	1	69.6	57.8	4.7
μb BCA	4	37.8	19.6	5.8
μb BCNA	3	-	137.9	8.7
μb ACK	32	11 279.4	47.6	8.8
VectorAdd	3	67.9	63.3	1.0
HBM2				
μb BCA	1	145.5	83.7	8.4
μb BCA	4	151.2	83.7	8.6
μb BCNA	3	-	118.8	13.9
μb ACK	32	4910.8	78.1	14.7
VectorAdd	3	9.8	83.7	8.7

In Wang's case, the errors come from an incomplete support of all LSU modifiers and not fully including the memory features (bandwidth, frequency, row misses, ...), unlike in this study.

On the other hand, the HLScope+ model for Xilinx devices considers memory bound applications where the estimation is primary affected by DRAM bandwidth. HLScope+ requires a board characterization to compute the controller overhead (Tco) [41]; this parameter is different for each benchmark because Tco varies with access type; this study uses Tco =2.5 ns for $\#lsu > 3$, and Tco =0 ns in other cases.

The two state-of-art models compared only support aligned and random access, but as this study shows, the memory strategies go one step further using HLS tools combining and modeling GMI and DRAM behavior.

In addition, note that Wang and HLScope+ do not adapt well to memory changes and only cover DRAM, unlike the proposal in this study that supports both.

8 Conclusions

As in other HPC processors, memory in FPGAs is one of the most critical aspects of system performance. This paper proposes an analytical model that identifies the main parameters that control the total execution time when the kernel-pipeline saturates memory bandwidth, a common situation for HPC applications. Specifically, the model determines the memory saturation through the relationship with memory occupation and kernel frequency and accurately estimates the kernel execution time without a time-consuming synthesis process, helping programmers and HLS tools to design and anticipate performance without extensive exploration processes, as used in other studies.

The model stems from a detailed study of the generated RTL code, instantiated IPs, and FPGA architecture without loss in flexibility that is demonstrated with two DRAM technologies: DDR4-1866 and 3D-stacked HBM2.

The results show the model has an average error of 11.4% for DDR4 and 10.4% for HBM2. Errors above average are directly associated with kernel frequency limitations in the compilation process. Compared with two state-of-the-art models, mainly focused on computing, the proposed model at least halves the error and shows adaptability to two technologies and memory frequency variations, unlike other proposals. Our future work aims to integrate this type of model into scheduling policies of heterogeneous systems, where predicting performance before launching a kernel can make a difference, helping to achieve higher performance and energy efficiency.

Acknowledgment

This work was supported by MINECO/AEI/ERDF (EU) (grant PID2019-105660RB-C21 / AEI / 10.13039/501100011033), Aragón Government (T58_20R research group), ERDF 2014-2020 "Construyendo Europa desde Aragón", and Santander-UZ grants program.

References

[1] S. M. Trimberger, "Three ages of fpgas: A retrospective on the first thirty years of fpga technology," *Proceedings of the IEEE*, 2015.

[2] B. C. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, 2020.

[3] Q. Gautier, A. Althoff, Pingfan Meng, and R. Kastner, "Spector: An opencl fpga benchmark suite," in *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 141–148.

[4] Y. Liang, S. Wang, and W. Zhang, "Flexcl: A model of performance and power for opencl workloads on fpgas," *IEEE Transactions on Computers*, vol. 67, no. 12, pp. 1750–1764, 2018.

[5] A. Verma, A. E. Helal, K. Krommydas, and W.-c. Feng, "Accelerating Workloads on FPGAs via OpenCL: A Case Study with OpenDwarfs," *Computer Science Technical Reports*, 2016.

[6] Intel, "Intel FPGA SDK for OpenCL Pro Edition: Getting Started Guide 19.1," 2019.

[7] H. R. Zohouri, N. Maruyamay, A. Smith, S. Matsuoka, and M. Matsuda, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, vol. 2016, no. November, p. 35, 2016.

[8] Z. Wang, B. He, W. Zhang, and S. Jiang, "A performance analysis framework for optimizing opencl applications on fpgas," in *HPCA*, 2016, pp. 114–125.

[9] Y. K. Choi, P. Zhang, P. Li, and J. Cong, "HLScope+,: Fast and accurate performance estimation for FPGA HLS," in *ICCAD*, 2017.

[10] J. Gómez-Luna, I. El Hajj, V. Chang, Li-Wen Garcia-Flores, S. Garcia de Gonzalo, T. Jablin, A. J. Pena, and W.-m. Hwu, "Chai: Collaborative heterogeneous applications for integrated-architectures," in *ISPASS*. IEEE, 2017.

[11] J. Jiang, Z. Wang, X. Liu, J. Gómez-Luna, N. Guan, Q. Deng, W. Zhang, and O. Mutlu, "Boyi: A systematic framework for automatically deciding the right execution model of OpenCL applications on FPGAs," *FPGA 2020*, 2020.

[12] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: A high-level performance analysis tool for fpga-based accelerators," in *DAC*, 2016, pp. 1–6.

[13] J. Zhao., L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "Comba: A comprehensive model-based analysis framework for high level synthesis of real applications," *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 430–437, Nov 2017.

[14] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "Performance Modeling and Directives Optimization for High-Level Synthesis on FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 7, 2020.

[15] H. R. Zohouri and S. Matsuoka, "The Memory Controller Wall: Benchmarking the Intel FPGA SDK for OpenCL Memory Interface," *2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pp. 11–18, 2019.

[16] G. Csordas, M. Asiatici, and P. Ienne, "In search of lost bandwidth: Extensive reordering of DRAM accesses on FPGA," *2019 International Conference on Field-Programmable Technology, ICFPT*, 2019.

[17] S. W. Nabi and W. Vanderbauwhede, "FPGA design space exploration for scientific HPC applications using a fast and accurate cost model based on roofline analysis," *Journal of Parallel and Distributed Computing*, 2016.

[18] B. da Silva Gomes, A. Braeken, E. D'Hollander, and A. Touhafi, "Performance and resource modeling for fpgas using high-level synthesis tools," in *Symposium ParaFPGA 2013, Parallel Computing with FPGAs*, vol. 25. IOS Press, 2014, pp. 523–531.

[19] C. Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016.

[20] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of fpgas and gpus," in *FCCM*, April 2018, pp. 93–96.

[21] S. W. Nabi and W. Vanderbauwhede, "Mp-stream: A memory performance benchmark for design space exploration on heterogeneous hpc devices," in *IPDPSW*, 5 2018.

[22] H. Choi, J. Lee, and W. Sung, "Memory access pattern-aware dram performance model for multi-core systems," in *ISPASS*, 4 2011.

[23] Intel, "Detecting Memory Bandwidth Saturation in Threaded Applications," 2010.

[24] J. Reinders, B. Ashbaugh, A. Bader, J. Brodman, J. Hammond, M. Kinsner, J. Pennycook, R. Schulz, and J. Sewall, *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL James*. Apress open, 2020.

[25] Intel, "External Memory Interface Handbook Volume 3: Reference Material," 2017.

[26] Intel, "Intel High Level Synthesis Compiler Pro Edition: Reference Manual," 2019.

[27] H. Zheng and Z. Zhu, "Power and performance trade-offs in contemporary dram system designs for multicore processors," *IEEE Transactions on Computers*, vol. 59, no. 8, pp. 1033–1046, 2010.

[28] Z. Jin and H. Finkel, "Optimizing an atomics-based reduction kernel on opencl fpga platform," in *IPDPSW*, 2018, pp. 532–539.

[29] Intel, "Intel® Stratix® 10 TX Product Table," 2019.

[30] Intel, "Intel® Agilex® I-Series SoC FPGA Product Table," 2019.

[31] S. Li, D. Reddy, and B. Jacob, "A performance and power comparison of modern high-speed dram architectures," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '18. Association for Computing Machinery, 2018, p. 341–353.

[32] K. Kara, C. Hagleitner, D. Diamantopoulos, D. Syrivelis, and G. Alonso, "High Bandwidth Memory on FPGAs: A Data Analytics Perspective," *FPL*, 2020.

[33] Intel, "External Memory Interfaces Intel ® Stratix ® 10 FPGA IP User Guide," 2019.

[34] W. Wang, T. Dey, J. W. Davidson, and M. L. Soffa, "Dramon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead," in *HPCA*, 2014, pp. 380–391.

[35] Micron-Technology, "DDRA SDRAM MT40A2G4," 2015.

[36] "Monitor Insider", "HBM2 Deep Dive," 2016. [Online]. Available: http://monitorinsider.com/HBM.html

[37] T. D. Matteis, J. de Fine Licht, and T. Hoefler, "FBLAS: Streaming Linear Algebra on FPGA," *CoRR*, 2019.

[38] NVIDIA, "NVIDIA OpenCL SDK Code Samples," 2020. [Online]. Available: https://developer.nvidia.com/opencl

[39] M. A. Dávila-Guzmán, R. G. Tejero, M. Villarroya-Gaudó, D. S. Gracia, L. Kalms, and D. Göhringer, "A cross-platform openvx library for fpga accelerators," in *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2021, pp. 75–83.

[40] Xilinx Vivado, "Vivado Design Suite User Guide: High-Level Synthesis," pp. 1–120, 2017.

[41] K. O'Neal and P. Brisk, "Predictive modeling for cpu, gpu, and fpga performance and power consumption: A survey," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2018.

Maria Angélica Dávila-Guzmán has been a PhD student in the Department of Computer Science and System Engineering at the University of Zaragoza, Spain, since 2017. She received her Master's and Bachelor's degrees in Electronic Engineering from the University of Valle, Colombia, in 2010 and 2015, respectively. Her research interests lie in heterogeneous systems, high-level syntheses, and load balancing.

Rubén Gran Tejero graduated in Computer Science from the University of Zaragoza, Spain. He received his Ph.D. from the Polytechnic University of Catalonia (UPC), Spain, in 2010. Since 2010, he has been an Associate Professor at the Department of Computer Science and Systems Engineering, University of Zaragoza. His research interests include hard real-time systems, hardware for reducing worst-case execution time and energy consumption, efficient processor microarchitecture, and effective programming for parallel and heterogeneous systems. Dr. Gran Tejero is member of the Aragon Institute of Engineering Research (I3A) and the Spanish Society of Computer Architecture (SARTECO).

María Villarroya-Gaudó obtained her Ph.D. in 2005 at the Department of Electronics Engineering at the Autonoma University of Barcelona. She is an Associate Professor in Computer Architecture and Technology in the Department of Computer and Systems Engineering at the Universidad de Zaragoza. Her research interests include memory hierarchy and heterogeneous systems. Dr. Vilarroya-Gaudó is member of the Aragon Institute of Engineering Research (I3A), the Spanish Society of Computer Architecture (SARTECO).

Darío Suárez Gracia (S'08–M'12) received his PhD degree in Computer Engineering from the University of Zaragoza, Spain, in 2011. From 2012 to 2015, he was at Qualcomm Research Silicon Valley. Currently, he is an Associate Professor at the University of Zaragoza. His research interests include parallel programming, heterogeneous computing, memory hierarchy design, energy-efficient multiprocessors, and fault-tolerance. Dr. Suárez Gracia is a member of the Aragon Institute of Engineering Research (I3A),the Spanish Society of Computer Architecture (SARTECO) the IEEE, the IEEE Computer Society, the ACM, and the HiPEAC European NoE.