Analytical Model of Memory-Bound Applications Compiled with High Level Synthesis

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

***Abstract*—The increasing demand of dedicated accelerators to improve energy efficiency and performance has highlighted FPGAs as a promising option to deliver both. However, programming FPGAs in hardware description languages requires long time and effort to achieve optimal results, which discourages many programmers from adopting this technology.**

High Level Synthesis tools improve the accessibility to FPGAs, but the optimization process is still time expensive due to the large compilation time, between minutes and days, required to generate a single bitstream. Whereas placing and routing take most of this time, the RTL pipeline and memory organization are known in seconds. This early information about the organization of the upcoming bitstream is enough to provide an accurate and fast performance model.

This paper presents a performance analytical model for HLS designs focused on memory bound applications. With a careful analysis of the generated memory architecture and DRAM organization, the model predicts the execution time with a maximum error of 9.2% for a set of representative applications. Compared with previous works, our predictions reduce on average at least 2× the estimation error.

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

I. Introduction

The promise of faster and more energy efficient systems with the inclusion of heterogeneity faces several challenges, specially for systems including re-programmable hardware such as FPGAs [1]. Fortunately for programmers, the development of High Level Synthesis (HLS) improves programmability and productivity, because CPU and GPU languages such C, C++, or OpenCL can be used to describe FPGA hardware [2], [3].

Although HLS tools simplify programming, generating highly tuned code remains a challenge for several reasons. First, CPU and GPU optimization techniques are not always directly suitable for FPGA, and, second, bitstream generation takes a long time, preventing any "trial-and-error" optimization process. To address this issue, programmers can follow two alternatives. Either they write well-known code patterns from previous explorations [4], or they rely on pre-synthesis analytical models for estimating performance [3], [5]–[7]. These models analyze the RTL code generated by the HLS tools, the high-level code, or both, and often they require to instrument and run the code to obtain dynamic profiling information.

The High Performance Computing, HPC, domain represents an opportunity for HLS and FPGAs because HPC requirements include performance and energy efficiency, and, ideally, to reuse as much code as possible. Many HPC applications are memory-bound, which together with the HLS generated code, enables to propose simple yet effective analytical models to predict their execution time when running on FPGAs. Besides, previous models focus more on the compute part, or kernel pipeline, covering the Global Memory Interconnect (GMI) connecting the kernel pipeline with the off-chip DRAM memory, in less detail. For example, the error of two state-of-the-art analytical models [6], [7] multiplies by 3 when the DRAM changes and can be larger than 50% for accesses with data dependencies. In future systems, those errors could become more prevalent because DRAM scaling is slow with a growing data rate of 7% per year, compared with FPGA that grows capacity 48% per year [8], potentially increasing the number of memory bound applications.

The GMI is composed by multiple Load/Store Units (LSUs) which include coalescers to group requests into DRAM burst, and arbiters to order them. HLS tools generate the GMI architecture at an early stage in the tool flow, so combining information from the GMI and DRAM organizations, it is possible to build an analytical model that mainly requires static information. The model can be easily plugged into existing models for memory-bound applications, or, even, integrated into HLS tools to guide optimizations.

The contributions of this work are: a) A description of the generated memory interconnect of an HLS tool, b) An open-source available analytical model that estimates the execution time of memory bound applications[1], c) A set of experiments showing that the error of the model is below 9% for a set of representative applications.

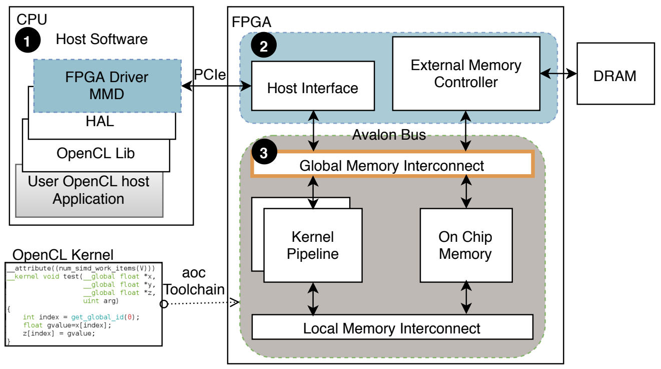
The rest of the paper is organized as follows. Section II describes the HLS flow and the GMI architecture. Section III introduces the model. Section IV presents the methodology. Section V comments on the results. Section VI discusses the related work and Section VII concludes.

II. HLS Flow for FPGA

Traditionally, HDL was the preferred tool to program FPGA devices, slowing its adoption by average programmers. Recently, HLS has evolved to a point where programming from languages such as C or OpenCL for FPGA becomes an easier task. In fact, the explicit parallelism of OpenCL offers many opportunities to exploit the pipeline parallelism inherent to FPGAs, making OpenCL a good language for FPGAs.

[1]The model with the experiment data are available at https://anonymous.4open.science/r/db707fea-264d-46bf-a6f8-f2cdf455d8d2/.

Figure 1 shows the main components of an OpenCL application compiled for an FPGA with the Intel OpenCL FPGA SDK. Without loss of generality, this flow is also representative for other toolchains. On the host side, ❶, the application communicates with the FPGA device through the Board Support Package[2] (BSP, in blue on the figure). The BSP implements the lower layers of the application stack such as the Memory-Mapped-Device (MMD) library, performing the basic I/O with the board and the PCIe communications. On the FPGA side, the BSP, ❷, provides support to communicate back with the host and with the device memory, DRAM . . .



```
__attribute((num_simd_work_items(V)))
__kernel void test(__global float *x,
                   __global float *y,
                   __global float *z,
                   uint arg)
{
    int index = get_global_id(0);
    float gvalue=x[index];
    z[index] = gvalue;
}
```

Figure 1. OpenCL FPGA main elements. Blue and brown colors represent the BSP and kernel logic, respectively.

From a programmer's perspective, the most important part is the kernel logic (in brown), ❸, which mainly corresponds to the compiled OpenCL kernel. In fact, programmers seldom need to generate a new BSP[3]. The compilation process consists of two main steps. First, a translator generates HDL code from the OpenCL, and then, a synthesis tool generates the bit-stream. The translator creates 4 blocks from the code: Local Memory Interconnect, On Chip Memory, Kernel Pipeline, and Global Memory Interconnect, being the last two the most critical from a performance point of view.

A. Kernel Pipeline

The kernel pipeline implements the whole 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 interconnects the pipeline with either the local or global memory. To exploit work-item parallelism, HLS tools implement very deep pipelines. Splitting up the processing into small pipeline stages also helps reaching high frequency, which is another key parameter for kernel performance besides pipeline length and initiation interval.

B. Global Memory Interconnect

The GMI communicates the Kernel Pipeline and the main DRAM memory. In OpenCL source code, each reference to a variable hold in the global memory constitutes a *global*

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

[3]Please also note that HLS tools would always require a BSP to compile an OpenCL kernel for supporting the aforementioned low-level tasks.

```
#define N 1024
int random_vector[N]={5,1023, 450, 100, ...}
__kernel void
test_patterns(  global int *restrict x,
                global int *restrict z,
                constant int *cn )
{   int i   = get_global_id(0);
    int j   = random_vector[i];
    int out = 0;  local int lmem[1024];
    //Code Snippet form Table I
    z[0]    = out;
}
```

Listing 1. OpenCL Code for access patterns in Table I

access. Since global accesses are the main source of kernel stalls, the GMI implements several strategies to maximize DRAM throughput and kernel pipeline flow. Internally, the GMI architecture, as other hardware memory interfaces from Intel [9], has two main components: LSUs, which track in flight memory operations, and arbiters. There are two independent round-robin arbiters ordering read and write accesses.

Depending on the access pattern, Intel FPGA SDK [10] has defined 5 LSU types: two for the Local Memory Interconnect (Constant-Pipelined and Pipelined) and the rest for the GMI: Burst-Coalesced, Prefeching, and Atomic-Pipelined. To understand the access pattern of each LSU, Listing 1 and Table I 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 DRAM, and 3) Atomic, which serializes the operation and guarantees atomicity. Please note that each one of these LSU features expose an increasing hardware complexity. Each *global access (GA)* stated in the source code can be translated into one or several LSUs, as Section III describes.

Each LSU type provides a different maximum bandwidth, being the burst-coalesced (BC) with aligned modifier the most efficient type because it maximizes DRAM effective-utilization. Figure 2 shows a read operation generated by a BC LSU. Each LSU has a coalescer unit that tries to group continuous memory address into a single burst DRAM operation. Eventually, the read arbiter dispatches this operation to the FIFO into the Avalon Interconnect in order to issue a DRAM access to the Memory Controller IP through the Avalon Bus. The benefits come from the DRAM organization [11] because during a read operation at least 3 commands are required: precharge (PRE), activate (ACT), read out (RD). PRE opens a row in every bank, then, ACT opens a row in a particular bank, and RD read 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 defines the maximum number of burst requests at compilation time, because contiguous access to memory enables to hide the overhead of PRE/ACT commands.

In burst coalesced LSU, three limits trigger a request to the DRAM: 1) Burst_cnt bus, that usually corresponds to memory page size, 2) Maximum number of threads allowed to be coalesced and 3) Time out to minimize stalls in the kernel

Table I. GMI LSU Types and their modifiers for Intel FPGA SDK, the codes snippets are from Intel FPGA Guides [10].

LSU Type	Description	Pipelined	Burst	Atomic	Code Snippets[a]
Burst-Coalesced[b]	Request are group into a set of DRAM burst				
Aligned	Index is contiguous and aligned to page size	✔	✔	✘	`out = x[i];`
Non_Aligned	Index has a modifier non-aligned to page size	✔	✔	✘	`out = x[3*i+1];`
Write_ACK	Index to access has dependencies	✔	✔	✘	`out = x[j];`
Cache	Index have repetitive dependencies	✔	✔	✘	`for (uint k=0; k<N; k++)` `z[N*i+k] = x[k];`
Prefetching	Compiled as Burst-Coalesced Aligned	✘	✔	✘	
Constant-Pipelined	Read from a constant cache	✔	✘	✘	`z[i] = cn[i];`
Pipelined	Requests are submitted immediately	✔	✘	✘	`out= lmem[li - i];`
Never-Stall	Connects the LSU without arbitration	✔	✘	✘	`lmem[li] = x[i];`
Atomic-Pipelined	For atomic operations	✔	✘	✔	`atomic_add(&x[0], 1);`

[a] Each code snippet corresponds to line 10 in listing 1.
[b] The burst-coalesced type has four modifiers affecting its organization.

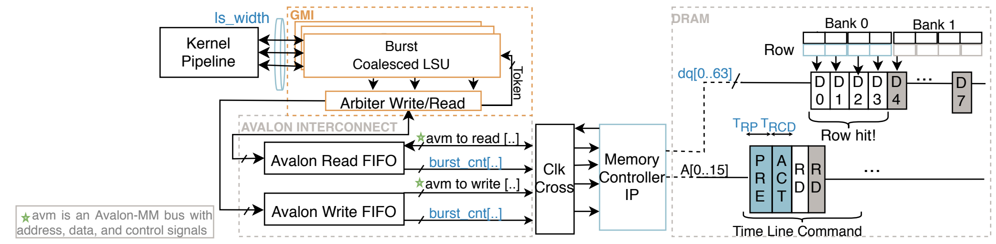


Figure 2. Simplified model of a read operation with LSU Burst-Coalesced modifier. The parameter names in blue are used in the model.

pipeline when the consecutive requests can not be coalesced. The compiler can modify this LSU depending on the memory access pattern and other attributes [10]; e.g., in the case of data dependencies, the compiler infers a Write-ACK LSU with a work-item level coalescer.

In a *Prefetching* LSU, the behavior is similar to the burst coalesced, but anticipating a large amount of data. For write operations, it uses a burst coalesced non-aligned LSU structure. In highend FPGAs, the compiler generates a Burst Coalesced LSU with given Intel SDK code.

The last LSU for GMI is the *atomic-pipeline*, Intel provides limited support only for 32-bit integers and it is considered one of the most expensive functions in HLS.

C. Performance Estimation for FPGA

Both the Kernel Pipeline and the GMI have a huge impact on performance. Kernel performance has been modeled to predict the execution time aiming at the automatization of the compilation process [3], [6], [7], but the memory component has been simplified, losing details that can expand the search space to optimize. This simplification, valid for old FPGA models, does not apply for newer models because the kernel resources' number has grown faster than the external memory; e.g., an Intel Stratix 10 reaches 9 TFLOPS while the newer Intel Agilex reaches 20 TFLOPS, although DRAM has improved from DDR4 @ 1333 MHz / 2666 Mbps to DDR4 @ 1600 MHz / 3200 Mbps or DDR5 @ 2100 MHz / 4400 Mbps, namely, kernel performance has doubled, meanwhile the memory bandwidth has not [12], [13]. Comparing the on-chip memory and the external memory, the external memory is more critical because the throughput of the DRAM is 380× worst than that of the embedded memory, and its size is 80× larger [12]. Therefore, a more accurate model of memory becomes desirable to estimate performance for the latest FPGA models.

III. Analytical Model

In memory bound kernels, the execution time of memory bound kernels is dominated by the GMI organization and the delay of the external DRAM memory.

In the kernel source code, a *global access (GA)* is translated into one or several LSUs at the GMI. The compiler determines the proper type of LSU for each *GA* according to a static analysis. As described in Section II, LSUs are part of the GMI that is modelled in this section. This model just relies on information available up to the translation phase (OpenCL to Verilog) providing accurate execution-time estimations for memory bound kernels without the long delays of the full compilation process.

Table II summarizes the input parameters to the model with their respective sources which are described below:

1) Report: Html file generated in intermediate compilation using *aocl -rtl*. It shows the kernel basic blocks and the types of the LSUs.
2) Verilog: These files instantiate the parameters of IPs and show stop conditions of memory controller. These files are generated with *aocl -rtl*.
3) Users: For dynamic loops, users are required to provide the iteration limit, since it is not available at compile

Table II. Variable descriptions, if possible, names come from the HLS tools

Variable	Definition	Source
$\#lsu$	Number of load/store units	Report
ls_width^i	Memory width of i LSU [bytes]	Report
$burst\_cnt^i$	Size of Avalon $burst\_count$ port param:BURSTCOUNT_WIDTH	Verilog
max_th^i	Maximum threads in a burst param:MAX_THREADS	Verilog
δ	Address stride of memory access	User
ls_acc^i	Number of access of each LSU	User
ls_bytes^i	Bytes of a single ls_acc	User
f	Kernel vectorization $SIMD \cdot Unroll$	User
dq	Memory data width [bytes]	Datasheet
bl	Memory burst length	Datasheet
f_mem	Memory frequency [Hz]	Datasheet
T_{RCD}	Row activation time [s]	Datasheet
T_{RP}	Precharge row miss time [s]	Datasheet
T_{WR}	Time to recovery from Write[s]	Datasheet

^{n} For every i LSU in a maximum of $\#lsu$

time. Please note that the user information could be automatically inferred by a compiler pass.

4) Datasheets: The DRAM datasheets provide the timing and the organization of DRAM memory chips.

To begin with, let T_{exe} be the execution time estimated as the sum of minimum time, T_{ideal}, of every transaction from every LSU plus an overhead time, T_{ovh}, which depends on the LSU type, as shows Eq. 1.

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

T^i_{ideal} only depends on maximum memory data transfer, and, hence, is the same for all LSU types, while T^i_{ovh} changes; e.g., when a memory access has a stride of δ^i, then coalescing LSUs do not use all data burst, which increases the amount of memory transactions. In other words, T^i_{ideal} corresponds to the minimum time for bringing every data which is the amount of bytes read by an LSU divided by the DRAM bandwidth, as shown in Eq. 2, with $bw\_mem = dq \cdot 2 \cdot f\_mem$ (The 2 corresponds to DRAM's double-rate).

$$T^i_{ideal} = \frac{ls\_bytes^i \cdot ls\_acc^i}{bw\_mem} \tag{2}$$

A. Burst-Coalesced LSU

In a memory bound application, the Avalon FIFO needs to be full of requests to maintain the DRAM with high occupancy. When the kernel pipeline does not make enough requests to fill the memory burst request before time out, the memory will fall down in low occupancy. In this state, the kernel is compute bound which has already been covered in previous works [6], [7].

Another cause of low memory occupancy is when the ls_width does not have enough data to allow the maximum burst size because ls_width depends on the vectorization factor (f) as well. To evaluate this condition, the relation between the number of kernel request to the LSU (ls_width) and the amount of data that DRAM can fulfil ($dq \cdot bl$) is calculated as shown in Eq. 3. Let K^i_{lsu} be the influence of each type of LSU modifier. When this condition is fulfilled, the kernel is defined as memory bound.

$$\text{Kernel Bound} \Rightarrow \begin{cases} \text{Memory high-occupancy} & \sum_{i=1}^{\#lsu} \frac{ls\_width^i}{dq \cdot bl \cdot K^i_{lsu}} \geq 1 \\ \text{Compute} & \text{otherwise} \end{cases} \tag{3}$$

Once the kernel is defined as memory bound, T_{exe} can be calculated with Eq. 1. To achieve T_{ideal} is possible for contiguous memory addresses, this type of access hides PRE/ACT latencies, as was shown in Fig. 2. Furthermore, bank-interleaving memory controller can completely hide opening new banks [14] until the $\#lsu$ is less than two. When the $\#lsu$ increases, this forces the DRAM to open a new row, adding a time overhead T_{ovh}.

The T^i_{ovh} will be proportional to DRAM latency of opening a new page, given by row miss commands (T_{row}). These can be calculated by the amount of times that an i LSU have to open a new row which depends of the amount of burst transactions, with a size of $burst\_size$, required to request the total bytes ($ls\_acc \cdot ls\_bytes$), formulated as in in Eq. 4. It should be noted that latency of LSU and the amount of data in a the Avalon FIFO would hide the kernel latency, for this reason only the DRAM latency is considered.

$$T^i_{ovh} = \begin{cases} 0 & \#lsu < 2 \\ \frac{ls\_acc^i \cdot ls\_bytes^i}{burst\_size^i} \cdot T_{row} & \text{otherwise} \end{cases} \tag{4}$$

The $burst\_size$, T_{row} and K_{lsu} estimations for each LSU modifier are analyzed in the subsections III-A1 to III-A3.

1) Burst Coalesced Aligned LSU: This modifier is generated when all the kernel requests are contiguous memory addresses aligned to page size.

Two kind of IPs have been detected in this type of access, one is the "simple" whether the program only has one LSU and every kernel request is sent directly to the memory controller, without FIFO registers. This simplifies the hardware and maximizes the memory throughput. With more LSUs the complete architecture is generated as was shown in Fig. 2

To calculate the memory constrain, the value of $K_{lsu} = \delta$ means that one burst operation is executed per cycle, but it is limited by strides.

The next step is the estimation of the DRAM $burst\_size$. DRAM sets the minimum burst transaction size to $dq \cdot bl$, but it can transfer multiple consecutive burst for the same open row giving Eq. 5. $burst\_cnt^i$ represents the binary logarithm of the transaction size, as shown in Fig. 2).

$$burst\_size^i = 2^{burst\_cnt^i} \cdot dq \cdot bl \tag{5}$$

Finally, to estimate T_{row} is not trivial because the controller can overlap commands due to reordering strategies and the page policy [15], then, this model takes into account the inter-command delay for row buffer misses [7] using ACT/PRE latencies, as Eq. 6 shows. The command sequence PRE, ACT, read and write are considered with the same minimum timing.

$$T_{row} = T_{RCD} + T_{RP} \tag{6}$$

2) Burst Coalesced Non-Aligned LSU: Both Aligned and Non-Aligned LSUs try to coalesce multiple thread request in a single burst command; however, the δ stride of Non-Aligned adds a new trigger for memory request, the number of threads, max_th, that have been launched and coalesced.

Eq. 7 calculates this constrain, named max_reqs, representing the maximum size of a DRAM burst-coalesced request. During the assembly of a request, two conditions can trigger the request issue towards the memory, either the amount of requested data equals a DRAM page, or the number of coalesced threads have reached max_th.

When a coalescer is assembling a request, either the request occurs when the amount of data requested is equals to a DRAM page or when the number of coalesced threads have reached max_th. This limit is affected by δ, effectively reducing the effective burst request. In the other case, the δ fraction of ls_width is the effective burst size as Eq. 8 shows. Please note that ls_width should be bounded by DRAM page size.

$$max\_reqs^i = \frac{max\_th \cdot ls\_width^i}{\delta + 1} \tag{7}$$

$$burst\_size^i = \begin{cases} \frac{max\_reqs^i}{\delta} & max\_reqs^i \leq 2^{burst\_cnt^i} \cdot dq \cdot bl \\ \frac{ls\_width^i}{\delta} & \text{otherwise} \end{cases} \tag{8}$$

Finally, the parameters K_lsu and T_{row} are the same ones as for burst coalesced aligned LSU.

3) Burst Coalesced Write-Acknowledge LSU : When data dependencies occur, the compiler generates a write acknowledgement signal to guarantee the correct ordering of accesses [10]. Therefore, the burst size equals the aligned case from Eq. 5, K_lsu equals 1, 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 Eq. 9 shows.

$$T_{row} = T_{RCD} + T_{RP} + T_{WR} \tag{9}$$

B. Atomic-pipeline LSU

Atomic-pipeline LSU only supports integer data type without burst commands; hence, its stride is always 1, $\delta = 1$. Every atomic operation executes a read and a write DRAM commands. For example, `atomic_add` from Listing 2 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 f operations atomically.

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

Listing 2. Atomic-pipeline add prototype function

Table III. Fixed variable value to evaluate the LSU model, all variables (Var.) are defined in Table II

Var.	Value	Var.	Value	Var.	Value
f_mem	933.3	dq	8	bl	8
T_{RCD}	13.5e-9	T_{RP}	13.5e-9	T_{WR}	15e-9

Eq. 10 shows the resulting T^i_{row}, including the two accesses, and T^i_{ovh}, depending on the f factor.

$$T^i_{row} = 2 \cdot (T_{RCD} + T_{RP}) + T_{WR}$$
$$T^i_{ovh} = \begin{cases} \frac{T^i_{row}}{f} & \text{val is constant} \\ T^i_{row} & \text{otherwise} \end{cases} \tag{10}$$

IV. Methodology

All the experiments have been run on an Intel Stratix 10 GX FPGA Development Kit with 2GB of DDR4 DRAM organized in a single DIMM and 4 memory banks. Table III shows the required DRAM parameters for the model, running at 1866MHz. The rest of parameters come from the intermediate compilation of the Intel FPGA SDK for OpenCL 18.1. For burst coalesced aligned and non-aligned LSU, δ variations are validated scaling the array accesses by δ. In the non aligned case, an offset argument is added to the scaled index.

To validate the model, two types of benchmarks are analyzed: first, a set of microbenchmarks, targeting each LSU type, where the vectorization factor f and the global access ($\#ga$) number vary using the Listing 3.

```
__attribute((num_simd_work_items(SIMD)))
__kernel void test_coalesced(
  __global const int *restrict x0,.., xn
  __global const int *restrict z   )
{
    int id = get_global_id(0);
    //code snippet from Table II for each LSU
}
```

Listing 3. OpenCL microbenchmark for LSU Coalesced Aligned

A second validation is performed in 9 different memory bound HPC benchmarks from the following sources: Intel FPGA SDK, Xilinx SDAccel, Rodinia FPGA [5], and FBLAS [16], which input channels were modified to fit the DRAM inputs.

The execution time is measured with aocl report, which compared with host measurements can have $\sim 8\%$ consistent difference. The atomic cases are measured with OpenCL events due to atomic LSU does not have dynamic counters implemented. Finally, the proposed model is compared with two state-of-the-art works [6], [7].

V. Model Validation

Two groups of experiments check the model accuracy: microbenchmarks, to dig into the keys of the model by sweeping the most common parameters such as $SIMD$ vector lanes, δ, or $\#lsu$, varying the number of global access (#ga), and, the second group being applications to showcase its effectiveness in real scenarios.

The model assumes that execution time depends on memory delay more than on kernel frequency for memory bound applications. To verify this claim, Fig. 3 shows the execution time for multiple sum reduction kernels with burst coalesced aligned LSUs (line 2 of Listing 4) varying $\#lsu$ and $SIMD$ vector lanes[4]. For memory bound kernels, encircled markers, F_kernel does not affect execution regardless $\#lsu$ and $SIMD$, because the memory delay dominates execution time. For non-memory bound kernels, uncircled markers, memory width, set by $SIMD$, affects performance more than F_kernel as well. Both results show the importance of early memory optimizations writing HLS code.

A. *Microbenchmarks*

For the sake of completeness, each LSU modifier is evaluated separately. The evaluation comprises the microbenchmark from Listing 3 with a body tuned to the LSU type and modifier. As shown in Listing 4, every loop body is based on sum reduction to easily change the number of GA, $\#ga$.

Fig. 4 shows the measured and estimated time for every type LSU under test varying $SIMD$ and $\#ga$. Empty bars represents non memory bound kernels, as detected with Eq. 3. For all the combinations, error remains bellow 15.7% for 75% of the benchmarks with a maximum error of 27.9%.

```
1     // Aligned
2     z[id] = x1[id] + ... + xn[id];
3
4      // Non-Aligned
5     z[3*id+1] = x1[3*id+1] + ... + xn[3*id+1];
6
7     // Write-Acknowledge
8     int id = rand[i]; //work item index
9     z[id] = x1[id] + ... + xn[id];
```

Listing 4. Sum Reduction microbenchmark for Burst Coalesced LSUs. Each modifier is run separately

1) Burst Coalesced Aligned LSU: Digging into each type, Fig. 4a compares the measured, T_{meas} and estimated, T_{exe} as the sum of T_{ideal} (dots) and T_{row} (lines), times for burst coalesced aligned, In this modifier, each global access generates one LSU ($\#ga$ equals to $\#lsu$). For all the cases, error remains below 10%, the probable source is the simplification of the DRAM model and the refresh time, which can reduce memory efficiency around 3.5% [14]. The experiment also evidences that the higher the $\#lsu$, the higher the T_{ovh}, so DRAM bandwidth reduces 26%, from 14.2 to 10.5 GB/s. For this reason, programming strategies such as *Array of Structures* reducing $\#lsu$ should be preferred.

The model shows a linear dependency to the stride parameter given by δ, increasing the number of times that the controller should access to DRAM as Fig 5a shows. Here the times are normalized to measured time with $\delta = 1$ and evidence how the estimation follows the linear tendency, marked with points. Notice that this LSU can not be generated with $\delta = 5$ because compiler does not detect the DRAM page size's alignment.

[4]Other LSU types produce the same results that are not shown for clarity.

2) Burst Coalesced Non-Aligned LSU: BCNA is depicted in line 5 of Listing 4 for a $K_{lsu} = 3$. Similarly to aligned modifier, in BCNA the global access is also supported by just one LSU. BCNA shows a larger error than BCA, between 4 and 21%, because the latency of BCNA coalescer has a large variance; e.g., the number of required address comparisons depends on the coalescer state. Please also note that error does not correlate neither with $SIMD$ vector lanes nor $\#lsu$, suggesting that the model correctly tracks parameter changes.

Also, for $SIMD$ and $\#lsu$ larger than 4 and 3, respectively, the number of threads in a burst significantly impacts execution time, which increases linearly and not exponentially as $SIMD$ does. This "max_th effect" can also be seen varying δ as Fig. 5b shows for $SIMD = 16$ and $\#lsu = 3$, times are normalized to $\delta = 1$. With $\delta = 7$ the max_th restriction appears optimizing the access that increases with strides. In comparison with an aligned LSU, the performance is reduced in median a 60% due address comparison increases and burst window is reduced to avoid long kernel stalls.

3) Burst Coalesced Write-Acknowledge LSU: The evaluation of this LSU uses the microbenchmark in Listing 3, with the code snippet in lines 7 to 9 of Listing 4.

A vector of constant values is generated by software with random values between 0 and 2048, reducing the probabilities of coalescing (2048 over 64 floats that can be coalesced in DRAM).

The SIMD vector lanes in the previous analyzed LSU affected the lsu_width, by contrast, with burst coalesced write acknowledge LSU the lsu_width remains constant. To increase the vectorization, the compiler generates so many LSU as the desired SIMD by each global access. The assumption is that every thread is accessing to different memory locations, controlling the memory consistency with the Ack signal. The Fig. 4c shows the comparison between the measured and estimated execution time.

The execution time is the worst of burst coalesced modifiers growing $24\times$ more than aligned LSU. The read operations show a stall on read until 98% with two LSU. 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.

4) Atomic-pipeline LSU: The evaluation of this LSU uses the microbenchmark in Listing 3, with the code snippet in Listing 5. To generate a single global access ($\#ga = 1$), $xn[id]$ has to be replaced by id, otherwise, each atomic operation generates its own global access to avoid coalescing.

```
    atomic_add(z[0], x[id]);
    ...
    atomic_add(z[n], xn[id]);
```

Listing 5. OpenCL microbenchmark for atomic-pipeline LSU

In general, atomic-pipeline LSU does not change the lsu_width, as burst coalesced does, making T_{ovh} the most significant component of this LSU. Fig 4d shows that execution time increases linearly with $\#ga$ and maximum error of 16% corresponding to unaccounted $5ns$ per atomic operation. This

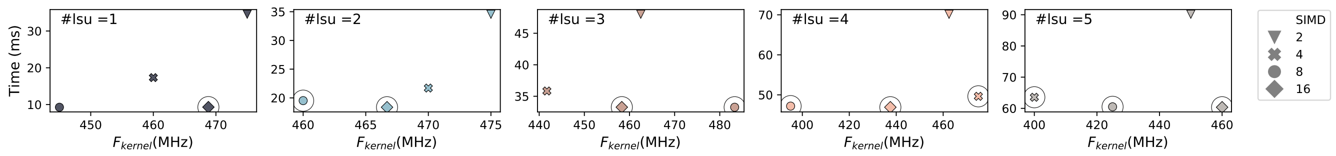
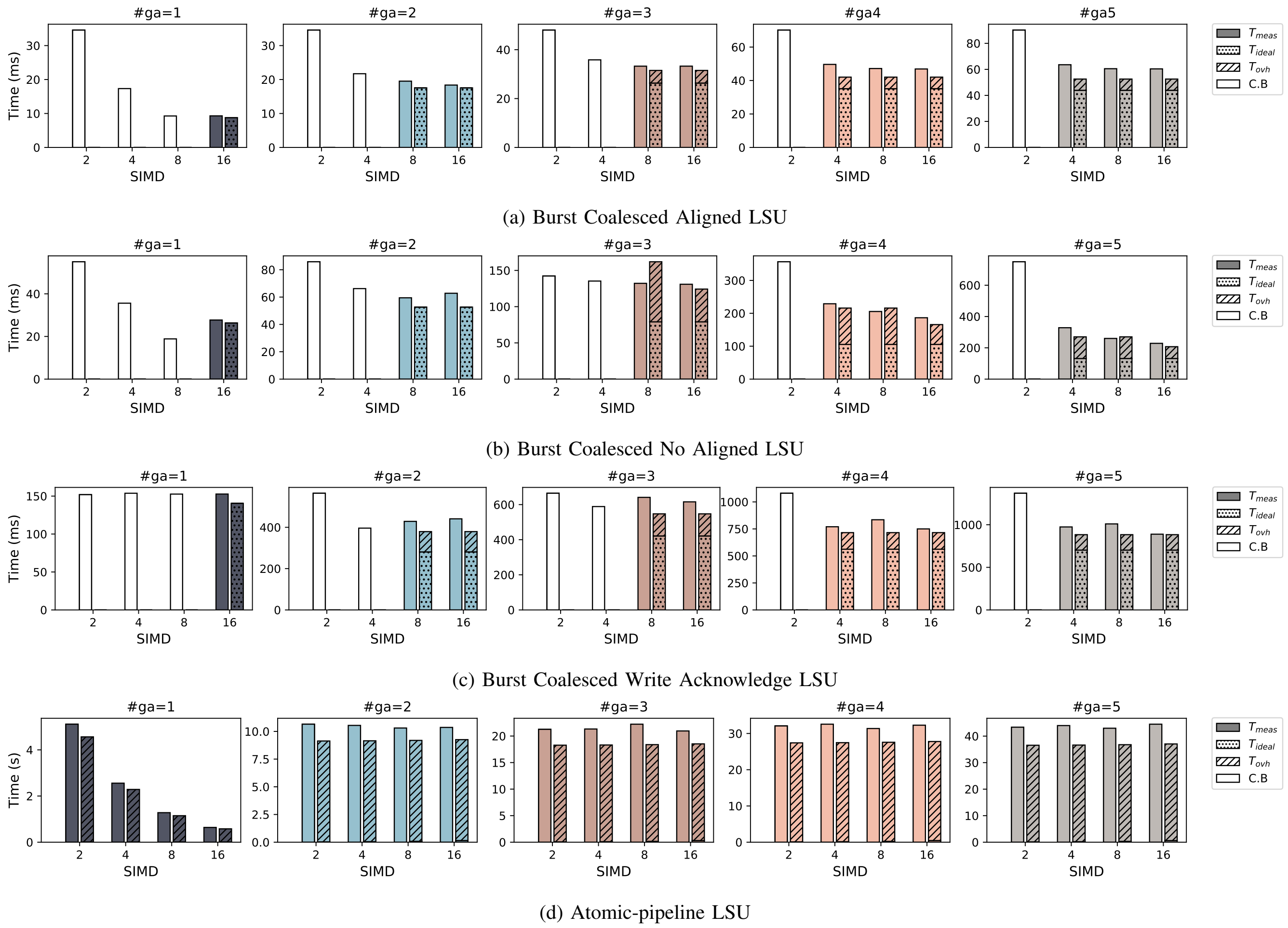


Figure 3. Execution time vs. kernel frequency in a burst coalesced aligned LSU varying $\#lsu$ and $SIMD$ vector lanes. Encircled markers correspond to memory bound kernels.


(a) Burst Coalesced Aligned LSU

(b) Burst Coalesced No Aligned LSU

(c) Burst Coalesced Write Acknowledge LSU

(d) Atomic-pipeline LSU

Figure 4. Measured (T_{Meas}) and Estimated ($T_{ideal} + T_{ovh}$) time for all the LSU's types varying $SIMD$ vector lanes and global access ($\#ga$). The bars with dots and lines represent T_{ideal} and T_{ovh}, respectively. Non memory bound kernels ($C.B$) are detected (empty bars) and not estimated.

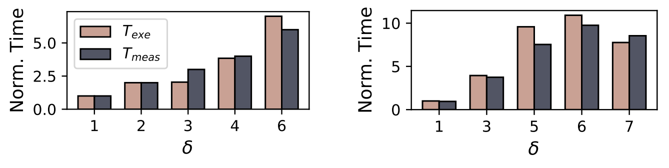

(a) Burst Coalesced Aligned LSU (b) Burst Coalesced Non aligned LSU

Figure 5. Measured (T_{Meas}) and estimated (T_{exe}) time are normalized to T_{Meas} in $\delta = 1$. The experiment varies δ with fixed values of $\#lsu = 3$ and $SIMD = 16$ in a) Burst coalesced aligned and b) Burst coalesced non-aligned LSUs

delay is near 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}).

To quantify the LSU impact on kernel performance, we analyze read stalls. For burst coalesced aligned and non-aligned, read stalls are under 20% because the coalescer partially hides the δ induced delay. Meanwhile, write ACK LSU shows stalls over 50% as the extra signalling serializes the requests. The atomic-pipeline modifier cannot be measured because profiling is unsupported, but it is safe to assume that stalls will be high due to atomicity requirements.

B. Applications

We validated the model with 9 memory bound applications, mixing single task and NDRange kernels with and without

Table IV. Kernel applications and estimated time .GMI-Global Memory Interconnect BCA-Burst Coalesced Aligned. BCNA-Burst Coalesced Non-Aligned. ACK-Burst Coalesced Write ACK. M- Measured. E- Estimated

Kernel	GMI	$\#lsu$	M.Time [ms]	E.Time [ms]	Error [%]
Dot [16]	BCA	3	60.2	64.5	7.3
FFT-1D [10]	BCA	2	9.5	8.8	7.3
nn [5]	BCA	2	157.5	172.1	9.2
ROT [16]	BCA	4	92.7	86.1	7.2
VectorAdd [10]	BCA	3	33.3	33.2	5.1
VectorAdd$\delta = 2$	BCA	3	67.9	63.0	6.5
Hotspot [5]	BCNA	3	9.7	8.8	8.7
Pathfinder [5]	BCNA	3	275.9	254.0	7.9
WM [17]	BCNA	2	59.8	55.8	6.6
NW [5]	ACK	4	1.4	1.4	4.0

channels. Table IV reports the measured and estimated time with the respective error. For all the applications, the relative error remains bellow 9.2% with an average value of 7.6%.

C. Comparison with other models

To compare the proposed model with two state-of-the-art models: Wang and HLScope+ [6], [7], we have manually computed their estimations for the microbenchmarks, with $f = 16$, and for the vectorAdd application. Unfortunately, the comparison of more applications is unfeasible because their dynamic profiling tools feeding the models are not publicly available. The tests are run with two BSPs with different DRAM frequency, 1866 and 2666 MHz.

In all, but one case, μb BCA, the error of this work is lower than that of Wang and HLSCope+ as Table V 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 V. Execution time estimated error; μb, BCA, BCN, and ACK refers to microbenchmark, burst coalesced aligned, burst coalesced non-aligned, and burst coalesced Write Ack, 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 BCN	3	-	71.1	4.0
μb ACK	2	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	2	11 279.4	47.6	8.8
VectorAdd	3	67.9	63.3	1.0

In Wang's case, the errors come from an incomplete support of all LSU modifiers and by not fully including the memory features (bandwidth, frequency, row misses, ...), which this work does. On the other hand, in HLScope+, for memory bound applications, the estimation is primary affected by DRAM bandwidth. Also, HLScope+ requires a board characterization to compute the controller overhead (Tco) [18]. This subsection uses $Tco = 2.5ns$ for $\#lsu > 3$, and $Tco = 0ns$ in other cases.

In addition, please note that Wang and HLScope+ do not adapt well to changes in memory, contrary to this work that is able to take them into account.

VI. Related Work

For many HPC applications running on FPGAs, memory is the main bottleneck [19], [20], contrary to these proposals that only focus on continuous and stride patterns, this work covers all possible patterns.

In the race to improve productivity with HLS in FPGA, performance modeling is a requirement. These models can be classified between those based on dynamic profiling [7], [21]–[24] an those based on static analysis [6]. Some predictive models have been compared for FPGA by [18], where only [7] covers external memory with the drawback that it requires on board characterization, and the memory interface is not portable to other boards [18]. In the same way, works, as [6] for OpenCL, have a coarse grain model that shows inaccuracies in the memory behavior and requires non-traditional HLS flow, some of the limitations of this work has been detected by [3]. Additionally, Liang *et al.* [3] improve models covering memory access patterns with a short CPU/GPU execution, but as some comparison shows, the memory controller makes difference in the performance [25]–[27], moreover CPU/GPU devices have a more sophisticated memory hierarchy that can hide DRAM latency. In [1], [28] they intent to guide programmers with analytical analysis of kernel, but the memory assumptions are limited without considering the memory interconnection, covered by our proposal.

VII. Conclusions

Compilation time represents an adoption barrier for HLS with FPGAs. This paper proposes an analytical model, estimating the execution time of memory bound applications accurately, so programmers can quickly guess the code performance without the time consuming synthesis. The model stems from a detailed study of the generated RTL-code, instantiated IPs, and FPGA architecture and condenses the factors causing the memory delay for all possible types of memory accesses, vectorization factors, strides, ... without loss in flexibility. Even, it supports changes in the board support package such as DRAM speed. By focusing on memory bound applications, the model does not depend on the kernel pipeline and can be easily plugged into existing compute bound oriented models and HLS tools. The model is publicly available, so it can be easily extended if required.

Our proposal has been carefully validated, and the obtained results show its accuracy predicting execution time. For 9 representative memory bound applications, the model error remains below 9%, and, compared with two other state-of-the-art model, it reduces the error by at least 2$\times$. Our future work aims to integrate such models into scheduling policies of heterogeneous systems, where predicting performance before launching a kernel can make a difference for achieving a higher performance and energy efficiency.

References

[1] H. M. Waidyasooriya, M. Hariyama, and K. Uchiyama, *Design of FPGA-based computing systems with openCL*, 2017.

[2] 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.

[3] 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.

[4] 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*, no. Section IV, pp. 1—-9, 2016.

[5] 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.

[6] Z. Wang, B. He, W. Zhang, and S. Jiang, "A performance analysis framework for optimizing opencl applications on fpgas," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 114–125.

[7] Y. K. Choi, P. Zhang, P. Li, and J. Cong, "HLScope+,: Fast and accurate performance estimation for FPGA HLS," vol. 2017-November, 2017, pp. 691–698.

[8] S. M. Trimberger, "Three ages of fpgas: A retrospective on the first thirty years of fpga technology," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 318–331, 2015.

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

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

[11] 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.

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

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

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

[15] 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 *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 380–391.

[16] T. D. Matteis, J. de Fine Licht, and T. Hoefler, "FBLAS: Streaming Linear Algebra on FPGA," *CoRR*, vol. abs/1907.07929, 2019.

[17] X. Vivado, "Vivado Design Suite User Guide: High-Level Synthesis," vol. 901, pp. 1–120, 2017. [Online]. Available: www.xilinx.com/products/design-tools/software-zone/sdaccel.html

[18] 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, pp. 763–768.

[19] 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*, pp. 1–13, 2016.

[20] B. Da Silva, A. Braeken, E. H. D'Hollander, and A. Touhafi, "Performance and resource modeling for FPGAs using high-level synthesis tools," in *Advances in Parallel Computing*, vol. 25. IOS Press BV, 2014, pp. 523–531.

[21] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: A high-level performance analysis tool for fpga-based accelerators," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[22] H. Mohammadi Makrani and et al., "Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 397–403.

[23] 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*, pp. 1–1, 2019.

[24] 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.

[25] 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)*, Nov 2016, pp. 1–8.

[26] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of fpgas and gpus," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2018, pp. 93–96.

[27] S. W. Nabi and W. Vanderbauwhede, "Mp-stream: A memory performance benchmark for design space exploration on heterogeneous hpc devices," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 5 2018, pp. 194–197.

[28] H. R. Zohouri, A. Podobas, and S. Matsuoka, "Combined spatial and temporal blocking for high-performance stencil computation on fpgas using opencl," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA 18. New York, NY, USA: Association for Computing Machinery, 2018, p. 153162.