Revista Colombiana de Computación
Vol. 25, No.2. July – Dec 2024, pp. 43-58
e-ISSN: 2539-2115, https://doi.org/10.29375/25392115.5276
Selected paper previously presented at the Latin America High Performance Computing Conference (CARLA 2023), an event held in Cartagena de Indias, Colombia, September 18-22, 2023.

REVISTA COLOMBIANA DE COMPUTACIÓN

Extending Ginkgo to Manage Reconfigurable Hardware-Based Kernels

Expansión de *Ginkgo* para administrar *kernels* reconfigurables basados en *hardware*

Alejandro Morales-Peña[1], Esteban Meneses[1,2]

[1] Costa Rica Institute of Technology, San José, Costa Rica
[2] National High Technology Center, San José, Costa Rica

alejandromp12.4@estudiantec.cr, emeneses@cenat.ac.cr

(Received: 6 February 2024; accepted: 18 June 2024, Published online: 31 December 2024)

Abstract. Although heterogeneous systems based on hardware accelerators are a trending topic in the HPC community, exploring the trade-offs of reconfigurable hardware-based ones in linear algebra libraries for high-performance systems, has not been deeply studied. Therefore, in this research, we aim to take advantage of FPGAs' reconfigurability, adaptability, and capacity to reduce power consumption to generate FPGA-based kernels in Ginkgo, a specialized high-performance linear algebra library for many-core systems. We generated 3 FPGA-based kernels for the CSR, SELLP, and SELL SpMV formats, and obtained speedups of at least 10x concerning CPU-based kernels. Furthermore, we demonstrated via a performance characterization study that FPGAs outperform general-purpose processors in terms of compute time.

Keywords: HPC, Ginkgo, FPGAs, SpMV.

Resumen. Aunque los sistemas heterogéneos basados en aceleradores hardware son un tópico de tendencia en la comunidad *HPC*, explorar las ventajas y desventajas de los basados en hardware reconfigurable en librerías de álgebra lineal para sistemas de alto rendimiento, no ha sido estudiado en profundidad. Por ello, en esta investigación, nuestro objetivo es aprovechar la capacidad de reconfiguración, adaptabilidad y reducción del consumo de energía de las *FPGAs* para generar *kernels* basados en *FPGAs* en *Ginkgo*, una librería especializada de álgebra lineal de alto rendimiento para sistemas multinúcleo. Generamos 3 *kernels* basados en *FPGA* para los formatos *CSR*, *SELLP* y *SELL SpMV*, y obtuvimos aumentos de velocidad de al menos 10 veces respecto a los *kernels* basados en *CPU*. Además, demostramos mediante un estudio de caracterización del rendimiento que las *FPGA* superan a los procesadores de propósito general en términos de tiempo de cálculo.

Palabras claves: Computación de Alto Rendimiento, *Ginkgo*, *FPGAs*, *SpMV*.

1 Introduction

Much of the work in developing linear algebra software for advanced architectures is motivated by the need to solve large problems in High-Performance Computing (HPC) systems. In fact, the linear algebra community has long recognized the importance of developing algorithms into software libraries that can help to leverage the power of supercomputers and other specialized systems to perform calculations involving matrices, vectors, and other data structures more efficiently (Dongarra & Blackford, 1996; Dongarra & Walker, 1995).

In addition to the wide range of HPC applications that rely on linear algebra and mathematical libraries to perform computations, it is well-known that solving linear systems is one of the most computationally and memory-intensive tasks in HPC (BSC, 2016; Anzt, et al., 2020). Thereupon, it becomes critical to find new ways to improve the performance of the linear algebra kernels and libraries in order to fulfill the resource demands of the applications built on top of them.

Current trends in the architecture of HPC systems are focused on providing heterogeneous execution environments to meet the ever-increasing demands for computing power (Bosch, et al., 2018). Heterogeneity comes in many different forms. One of the most important is employing specialized accelerators such as Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs) (Tsoi & Luk, 2010; Gao & Zhang, 2016; Steffenel, 2019). FPGA-based hardware accelerators have recently gained popularity due to their reconfigurability, adaptability to multiple tasks, ability to deliver better performance and power efficiency than general-purpose processors (CPUs), and even to compete with GPUs in terms of raw execution performance and power efficiency (Sommer, Korinth, & Koch, 2017; Podobas, 2014).

Thus, with the aim of clarifying the performance trade-offs of FPGA-based hardware accelerators in linear algebra parallel programs we explore the effects of using reconfigurable hardware (FPGAs) in Ginkgo (Anzt, et al., 2020), a specialized high-performance linear algebra library for manycore systems. Although Ginkgo already supports execution on OpenMP (2024) for multi-threading CPU-based kernels, and CUDA (NVIDIA Corporation, 2024), HIP (AMD, 2023), and SYCL (2024) for GPU kernels, there is no design focused on FPGAs accelerators. Therefore, by employing OpenCL (2024) in this study we extend Ginkgo to produce a brand-new set of FPGA-based kernels.

More in detail, we centered this initial approach on the sparse matrix-vector product (SpMV) kernels of Ginkgo and generated three new versions (not optimized) for the matrix formats CSR, ELL, and SELLP (Townsend, 2016). Afterward, by using six distinct dense matrices (categorized by their number of rows, ranging from 1000 to 16000), we conducted an endurance assessment to characterize the performance of the FPGA and CPU-based kernels. Lastly, in order to identify any improvement in terms of kernel execution time, we applied an analysis of variance (ANOVA) (Girden, 1992) to the experimental results and found that FPGA-based kernels are statistically different from CPU ones, and even we concluded that they can overcome the CPU-based ones at least 10x times (when using the maximum number of work items allowed by OpenCL). For this reason, we believe that implementing optimizations to the OpenCL designs will help to improve the kernels' efficiency and consequently Ginkgo's library as well.

We summarize the contributions of this research as follows:

- An understanding of the effects of reconfigurable hardware accelerators on the performance of linear algebra parallel programs;
- an extension to the Ginkgo library to support FPGAs as hardware accelerators of general-purpose multicore processors; and
- a brand-new alternative accelerator for Ginkgo that fosters the extraction of more computational power than general-purpose processors during computer simulations of linear algebra programs.

The remainder of this paper is organized into seven sections. Section 2 brings an overview of related works and contrasts them with the approach used for this research. The problem that this investigation aims to tackle is defined in section 3, together with the key concepts that relate to it. Section 4 details the algorithms generated for the FPGA-based kernels along with a chunking algorithm produced for the Ginkgo host code portion. In section 5, we describe the methodology implemented to extend Ginkgo and to drive the performance characterization of the FPGA and CPU-based kernels designed. Section 6 presents the outcomes of the experiments we carried out for the performance characterization. In section 7, we perform a deep analysis of the experimental results. Finally, in section 8 this paper is concluded and future directions for this work are provided.

2 Related Work

Within the existing state-of-the-art literature of linear algebra libraries for HPC systems, there are particular efforts focused on leveraging the benefits offered by reconfigurable hardware-based accelerators.

For instance, in Zhuo and Prasanna (2005), the authors propose an FPGA-based design for BLAS (Basic Linear Algebra Subprograms (Lawson, Hanson, Kincaid, & Krogh, 1979)) operations. The proposal guarantees very-low memory bandwidth requirements, exploits the memory hierarchy characteristic offered by FPGAs, and utilizes several of them (at the same time) for accomplishing the linear algebra operations involved in the problem under study.

A similar solution to Zhuo and Prasanna (2005), is presented in Sun, Peterson, and Storaasli (2007). The major difference is that the authors improve the implementation by (1) generating hardware that does

not need to change regarding the input matrix; (2) minimizing the initialization time; and (3) reducing I/O operations.

One of the more similar approaches to our work is fBLAS (De Matteis, de Fine Licht, & Hoefler, 2020), a flexible and customizable implementation of BLAS for FPGAs. It equally to our solution relies on HLS (high-level synthesis (Zhang, et al., 2008)) to favor the rapid development of numerical computations.

Another BLAS implementation is available in Kestur, Davis, and Chung (2012). It introduces a novel hardware-optimized sparse matrix representation called Compressed Variable-Length Bit Vector (CVBV), which reduces the storage and bandwidth requirements by up to 43% (on average 25%) compared to the compressed sparse row (CSR) format.

LAPACKrc (Gonzalez & Núñez, 2009) is another attempt that employs reconfigurable hardware as accelerators. It is a family of FPGA-based linear algebra solvers. LAPACKrc subsumes some of the LAPACK (Anderson, et al., 1999) and ScaLAPACK (Dongarra & Blackford, 1996) functionalities for linear algebra and incorporates sophisticated general direct and iterative sparse matrix solvers (that are at least two orders of magnitude faster than traditional solvers).

Generally speaking, the most significant advantage of these approaches in regard to our work is they exploit more deeply hardware capabilities and employ more than one FPGA, which boosts computing performance. On the other hand, in contrast to our work, none of these solutions treats linear operations as fundamental objects, and nor they provide the composability and extensibility of the Ginkgo library. Additionally, due to its design, Ginkgo can be easily integrated into programs intended to solve linear algebra operations, which accelerates the development process without sacrificing performance. Finally, in distinction to traditional libraries such as BLAS and LAPACK, which leave memory management to the user, Ginkgo allocates/deallocates its memory automatically, using the C++ "Resource Acquisition Is Initialization" (RAII, 2024) concept combined with the native allocation/deallocation functions of the executor (Anzt, et al., 2022).

3 Background

3.1 Computing SpMV in HPC Systems

Sparse matrix-vector multiplication is widely used in a variety of high-performance applications in science and engineering. Often, the SpMV operations are iterative or repetitive and consume a large percentage of the runtime of applications. Therefore, finding new ways to improve their performance is critical.

In its simplest form SpMV is the operation $y = Ax$, where A is an $M \times N$ matrix, x is a vector of length N, and y is a vector of length M. On the other hand, sparse matrices differ from dense matrices in that they contain mostly zeros, which makes the number of non-zeros (nnz) an important measurement of sparse matrices. In most datasets, nnz grows by $O(M)$ instead of $O(MN)$ (Townsend, 2016).

SpMV has several formats, but for this research, we focus on compressed sparse row (CSR), ELLPACK (ELL), and the sliced ELLPACK (SELLP) formats. Such formats representations are detailed in Figure 1.

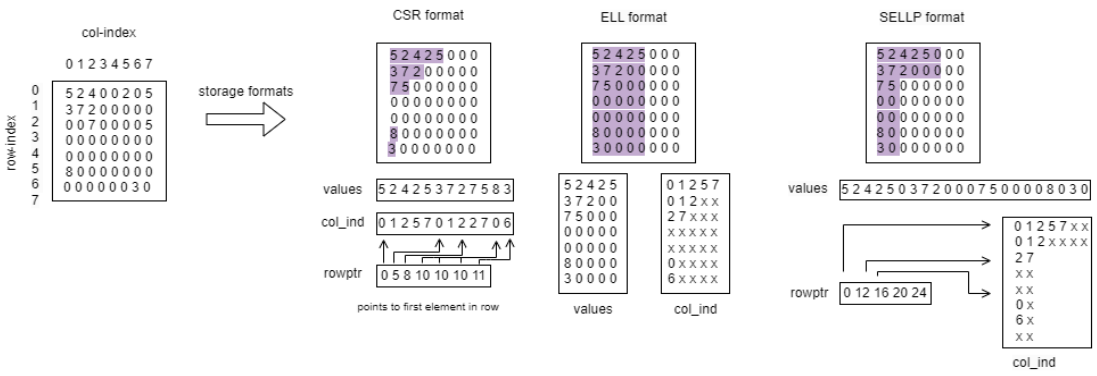


Figure 1. Dense and sparse matrix storage formats representations: CSR, ELL, and SELLP

3.2 Reconfigurable Hardware

Field-Programmable Gate Arrays (FPGAs) are pre-fabricated silicon devices that can be electrically programmed to become almost any kind of digital circuit or system (a basic FPGA architecture is shown in Figure 2). They are referred to as "field-programmable" because they provide the ability to reconfigure the hardware to meet specific use case requirements after the manufacturing process (Kuon, Tessier, & Rose, 2008).

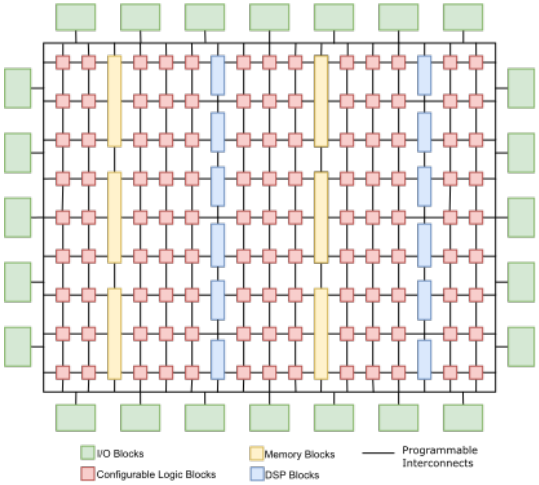


Figure 2. Basic FPGA architecture

FPGAs Programming. The user-defined logic in the FPGA is typically specified using a Hardware Description Language (HDL). The most common HDLs are VHDL and Verilog and extensions such as SystemVerilog. Unlike software programming languages that handle sequential instructions (such as C/C++), HDLs describe and define an arbitrary collection of digital circuits. Therefore, developers must understand digital electronics design, which includes understanding how the system is structured, how components run in parallel, meet timing requirements, and how to trade-off between different resources (Fang, Mulder, Hidders, Lee, & Hofstee, 2020).

More recently, High-level synthesis (HLS) tools such as Vitis HLS and OpenCL have overcome the problem described before by supporting software programmers with the feasibility of compiling standard languages such as C/C++ and higher-level hardware-oriented languages like SystemC into the Register-Transfer Level (RTL) designs. Moreover, apart from rendering the circuit itself, programming frameworks such as OpenCL provide tools for designing programs that run on heterogeneous platforms (e.g., a combination of CPU and FPGA) (Fang, Mulder, Hidders, Lee, & Hofstee, 2020).

3.3 Ginkgo Library

Ginkgo is a next-generation, high-performance sparse linear algebra library for multicore and manycore architectures. The library combines ecosystem extensibility with heavy, architecture-specific kernel optimization using the platform-native languages CUDA (for NVIDIA GPUs), HIP (for AMD GPUs), and OpenMP (for general-purpose multicore processors).

Ginkgo is built around two key design concepts. The first is concerned with the class and object-oriented design implemented for the linear operators, which aims to provide a user-friendly interface shared by all devices and linear algebra objects. The low-level device-specific kernels comprise the second main design concept, such kernels are optimized for the particular offloading device and use C++ features to generate high-performance kernels for various parameters (Anzt, et al., 2022).

SpMV Kernels. The sparse matrix formats provided by Ginkgo are COO, CSR, ELL, HYBRID, and SELLP. Additionally, Ginkgo provides high-performance conversion routines between the different matrix formats, enhancing their flexibility.

Iterative Solvers. The iterative solvers included in Ginkgo are the Krylov subspace methods: Conjugate Gradient (CG), Flexible Conjugate Gradient (FCG), Bi-Conjugate Gradient (BiCG), and its stabilized version (Bi-CGSTAB), Generalized Minimal Residual Method (GMRES), and the Iterative Refinement (IR) method. The library also features support for direct and iterative triangular solutions.

Preconditioners. The preconditioners offered by the library are Block Jacobi, ParILU, and ParILUT, as well as Incomplete Sparse Approximate Inverse preconditioners.

4 Reconfigurable Hardware-Based Kernels for SpMV Operations

For this research, we implemented 3 algorithms for solving SpMV computations along with a chunking algorithm produced to handle size constraints in OpenCL work items.

4.1 SpMV Kernels

The first 3 algorithms are SpMV kernels, for the CSR, ELL, and SELLP matrix formats. They are designed with OpenCL and launched via a wrapper code on Alveo/Xilinx FPGAs devices.

By using *get_global_id(0)* in the kernels, we exploited the parallelism provided by the FPGA device-offloading. Such a function indicates to the Vitis compiler to generate as many hardware blocks (HWBs) as possible to process that for loop. In these specific cases, we parallelize in the same order as the number of rows in the matrix under evaluation. Therefore, if the matrix comprises 3000 rows, the computation will be performed by 3000 HWBs (each HWB works with a specific row). The main drawback is that this solution is restricted by OpenCL constraints in the number of work units (4k).

The OpenCL kernels generated are detailed in Listing 1, Listing 2, and Listing 3.

Listing 1. OpenCL SpMV kernel for processing matrices in CSR format

```
__kernel void spmv_csr_kernel(int num_rows, __global const int *row_pointer,
__global const int *column_indices, __global const float *values, __global
const float *x, __global float *y)
{
  int row_id = get_global_id(0);
  if (row_id < num_rows) {
    float sum = 0.0;
    int row_start = row_pointer[row_id];
    int row_end = row_pointer[row_id + 1];
    for (int i = row_start; i < row_end; i++) {
      sum += values[i] * x[column_indices[i]];
    }
    y[row_id] = sum;
  }
}
```

Listing 2. OpenCL SpMV kernel for processing matrices in ELL format

```
__kernel void spmv_ell_kernel(int num_stored_elements_per_row, int a_stride,
int c_stride, int num_rows, __global const float *values, __global const int
*column_indices, __global const float *x, __global float *y)
{
  int gid = get_global_id(0);
  if (gid < num_rows) {
    float sum = 0.0;
    for (int j = 0; j < num_stored_elements_per_row; j++) {
      int indx = gid + (j * a_stride);
      int col_idx = column_indices[indx];
      if (col_idx != -1) {
        sum += values[indx] * x[col_idx];
      }
```

```
    }
    y[gid * c_stride] = sum;
  }
}
```

Listing 3. OpenCL SpMV kernel for processing matrices in SELLP format

```
__kernel void spmv_sellp_kernel(int block_size, int b_stride, int c_stride,
__global const float *values, __global const int *column_indices, __global
const int *slice_offsets, __global const float *x, __global float *y)
{
  int gid = get_global_id(0);
  int slice_id = gid / block_size;
  int row_id = gid % block_size;
  int slice_start = slice_offsets[slice_id];
  int slice_end = slice_offsets[slice_id + 1];
  float sum = 0.0;
  for (int j = slice_start; j < slice_end; j++) {
    int indx = row_id + (j * block_size);
    if (column_indices[indx] != -1) {
      sum += values[indx] * x[column_indices[indx] * b_stride];
    }
  }
  y[gid * c_stride] = sum;
}
```

4.2 Chunking Algorithm

The chunking algorithm was born due to the necessity of allowing FPGA-based kernels to handle matrices with more than 4k rows. Because, as detailed above in our implementations each row of the matrices under study corresponds to a work item in OpenCL. Unfortunately, OpenCL has the limitation of only being able to execute 4k work items at the same time. This means the kernels designed are not able to process in parallel more than that number of rows. The algorithm pseudo code is detailed in Algorithm 1.

Algorithm 1. Chunking algorithm for handling size constraints in OpenCL work items

```
max_allowed_elements = 4k
max_work_group_size = cl_kernel_work_group_size
num_elements_to_process = num_matrix_rows
work_group_size = min(max_work_group_size, num_elements_to_process)
global_offset = 0

IF (num_elements_to_process > max_work_group_size) THEN
  work_group_size = min(max_allowed_elements, max_work_group_size)
  num_elements_to_process_itr = work_group_size

  WHILE (num_elements_to_process > 0) DO
    run_kernel(global_offset, num_elements_to_process_itr, work_group_size)
    num_elements_to_process -= num_elements_to_process_itr
    global_offset += num_elements_to_process_itr
    work_group_size = min(max_work_group_size, num_elements_to_process)
    work_group_size = min(max_allowed_elements, work_group_size)
    num_elements_to_process_itr = work_group_size
ELSE
run_kernel(global_offset, num_elements_to_process, work_group_size)
```

5 Methodology

5.1 Ginkgo Project Extension

In order to adapt Ginkgo to support a new accelerator we advantage of one of its appealing features, which is the ability to run code (kernels) on a variety of device architectures transparently via the Executor class at its core. Briefly, the Executor class specifies the memory location and the execution space of the linear algebra objects and represents the computational capabilities of distinct devices (Anzt, et al., 2022). Currently, five executor types are provided: (1) CudaExecutor for CUDA-enabled GPUs; (2) HipExecutor for HIP-enabled GPUs; (3) OmpExecutor for OpenMP execution on multicore CPUs; (4) DpcppExecutor for Intel-enabled and other GPU devices; and (5) ReferenceExecutor for sequential execution on CPUs (used for correctness checking). Figure 3 depicts such architecture.

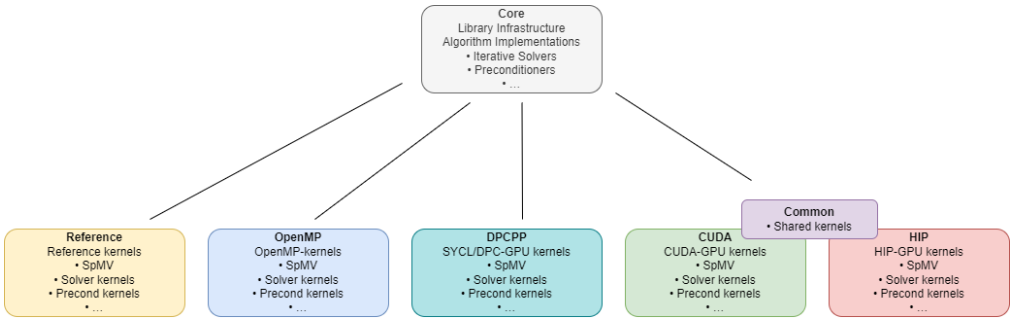


Figure 3. Ginkgo architecture representation. The design separates the core containing the algorithms from architecture-specific backends

Generally speaking we added a new Executor named OclExecutor (OpenCL-based) capable to run SpMV kernels, for the CSR, ELL, and SELLP matrix formats on Alveo/Xilinx FPGAs. We performed this in 3 phases. The first consisted of modifying the rest of the Ginkgo code to handle this new Executor and the prototyping kernels. In terms of parallel design specifics, we selected to run the whole kernel code in the FPGA and not only specific code regions. In the second phase, we care about the communication process between the OpenCL code (hardware side) and the host part. We dynamically mapped tasks to each OpenCL work item concerning the number of rows the matrix under analysis has (there is a work item for each matrix row and, consequently, a hardware section for each as well). Lastly, we generated the CL algorithms of the 3 kernels and incorporated an OpenCL wrapper into Ginkgo to; (1) load FPGA bitstreams; (2) invoke specific kernels; and (3) select the FPGA device to use.

5.2 System Description

The experiments conducted in this study used the computing resources provided by the Heterogeneous Accelerated Compute Clusters (HACC (2022a)) program promoted by AMD (2024), and maintained by the ETH Zurich (2024), a public university in Switzerland. Table 1 and Table 2 detail the system resources of the server where we ran our experiments (the server uses *Ubuntu 20.04.6 LTS* and runs the Linux kernel *5.4.0-155-generic*), and Table 3 lists all software versions used.

Table 1. System resources summary: host machine

Host machine summary	
Architecture	x86_64 Intel Xeon Processor
Processor	(Cascadelake) @ 3.2 GHz, 16 core(s), 16 Logical Processor(s)
RAM	128 GB
L3 cache	256 MB
SSD	300 GB
FPGA	Alveo U250 card

Table 2. System resources summary: FPGA device

FPGA summary	
FPGA Model	VU13P
Look-up tables (LUTs)	1728 K
Registers	3456 K
DDR memory	64 GB
DSPs	12228
UltraRAM	368.0 Mb
SRAM	54.0 MB

Table 3. Software libraries/program versions used in the experiments

Program/library name	Version
OpenMP library	3.0
OpenCL library	1.2
XRT library	2022.1
pthread library	3.2.0
v++	2022.1
gcc (C++14 compliant)	9.4.0
g++ (C++14 compliant)	9.4.0
make	4.2.1
cmake	3.16.3

5.3 Experimental Procedure

Experimental Configurations. Intending to be as much fair as possible in the experiments carried out for this work, we selected measuring the kernel execution time of the Reference and OpenCL Executors (we exclude the overhead time of moving data buffers from the host to the offloading device). Our study, on the other hand, consisted of running SpMV calculations using six distinct dense matrices (categorized by the number of rows), where the input vector used in all trials remained constant (a column vector filled with 1's). We ran 25 SpMV calculations for each matrix, Executor type, and kernel. In other words, we recorded a total of 750 executions. An important aspect is we ran two warmup runs before each set of trials, and after each SpMV calculation we performed a hot reset on the FPGA.

We summarize the testing environments in Table 4, and Table 5 recaps the matrices utilized (all matrices were taken from the SuiteSparse Matrix Collection (Davis & Hu, 2011)).

Table 4. Summary of the experimental setup

Kernel	Executor	Matrices Rows	Execs. per Matrix
CSR	…	…	…
SELL	OpenCL and Reference	1k, 2k, 4k, 8k, and 16k	25
SELLP	…	…	…

Table 5. Matrices utilized in the experiments

Matrix	Domain	Number of rows	Number of non-zeros
olm1000	Computational Fluid Dynamics Problem	1000	3996
olm2000	Computational Fluid Dynamics Problem	2000	7996
tols4000	Computational Fluid Dynamics Problem	4000	8784
windtunnel_evap2d	Computational Fluid Dynamics Problem	8256	109368
ex11	Computational Fluid Dynamics Problem	16614	1096948

Experiments Automation and Data Acquisition. By using scripting, we automated the experiment execution and data-collecting process. In general, the script takes care of launching each test environment and writing a report based on the information generated by the Ginkgo-based application (as it was progressing).

5.4 Performance Characterization Process

To find any improvement in kernel execution time, we utilized statistical concepts and performed an analysis of variance (ANOVA) on the collected experimental results, to finally elaborate conclusions based on those outcomes. Furthermore, we split the results into the following categories:

- Kernel type: executor, which runs the kernels.
- Kernel: matrix format employed.
- Matrix rows: number of rows of the matrix under test.

We also conducted a scalability study to determine the speedup per sample of the FPGA-based kernels concerning the reference kernels.

Finally, in order to reduce unnecessary system overhead, we restricted the server to just performing our tests while they were running.

6 Experimental Results

6.1 Scalability Study

The first experiment performed consisted of performing a scalability analysis of the results obtained. For it, we extracted the speedup measures of each kernel concerning the number of rows of the matrix under test (regarding our design each row corresponds to a work item in OpenCL, hence the number of work items is proportional to the number of rows in the matrix).

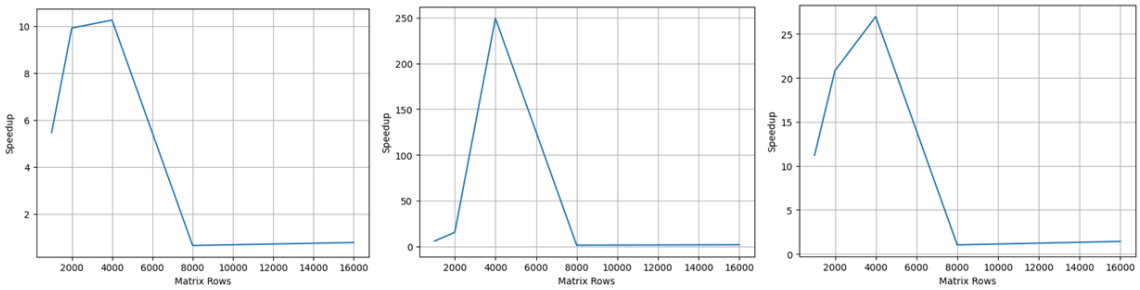
In Equation (1), we denote the mathematical expression used for the speedup (S) calculations. Where T_{seq} refers to the kernel execution time obtained when using the ReferenceExecutor, and T_{par} is the execution time of the OclExecutor (parallel version).

$$S = \frac{T_{seq}}{T_{par}} \tag{1}$$

The results are presented in the Table 6, and the speedup plots are detailed in Figure 4. Additionally, because for each matrix type, we recorded 25 values, we had to estimate the average value of the kernel execution time for processing each matrix for these metrics.

Table 6. Speedup measures of each kernel

Kernel	Speedup	Number of rows
CSR	5.467808	1000
CSR	9.918942	2000
CSR	10.256257	4000
CSR	0.656187	8000
CSR	0.782288	16000
ELL	5.784231	1000
ELL	15.343091	2000
ELL	249.278149	4000
ELL	1.308149	8000
ELL	1.782723	16000
SELLP	11.215276	1000
SELLP	20.818575	2000
SELLP	26.969241	4000
SELLP	1.029992	8000
SELLP	1.424910	16000



(a) CSR. (b) ELL. (c) SELLP.

Figure 4. Speedup plots of the CSR, ELL, and SELLP kernels

6.2 Analysis of Variance

To conclude our performance characterization study, we developed a formal analysis of variance in which the ANOVA statistical approach is implemented to separate the observed variance data into several components (respecting the Executor and kernel types, as well as the number of rows processed). We summarize the main results ANOVA results (p-values) in Table 7, Table 8, and Table 9. Furthermore, in Figure 5 we present the plots associated with the p-values outcomes we obtained. These particular results are crucial because they show the effects of the FPGA-based kernels compared to the reference kernels of Ginkgo.

Table 7. Executors contrasting via p-values (*alpha* = 0.05)

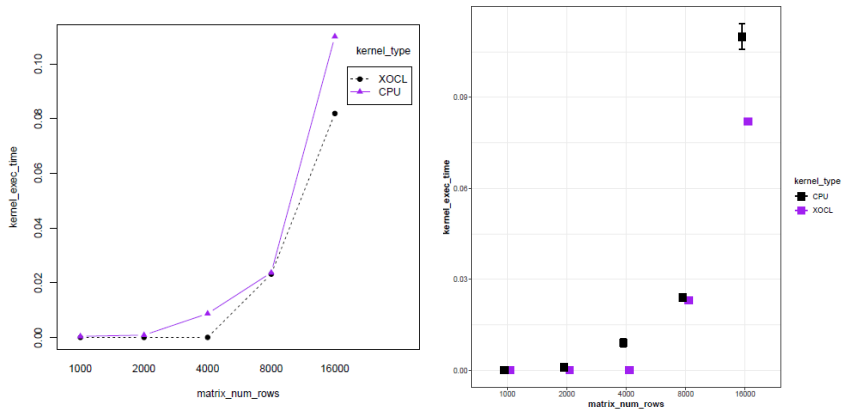
Contrast group	p-value
OclExecutor vs. ReferenceExecutor	<.0001

Table 8. Contrasting: Executors, and kernels, via p-values (*alpha* = 0.05)

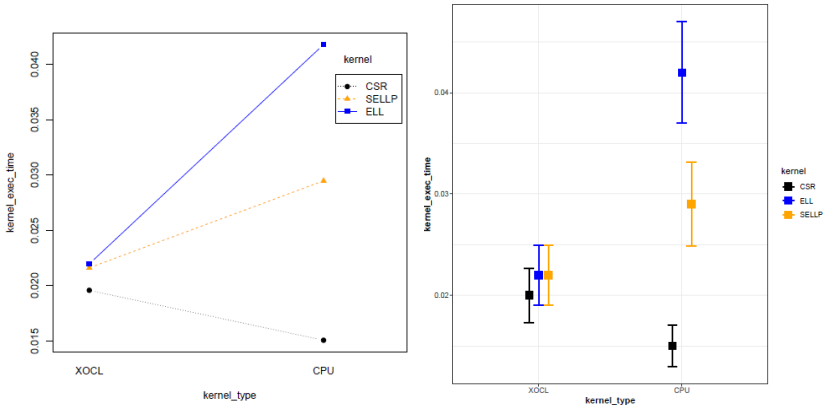
Contrast group	p-value
OclExecutor - CSR vs. ReferenceExecutor - CSR	<.0001
OclExecutor - CSR vs. OclExecutor - SELLP	<.0001
OclExecutor - CSR vs. ReferenceExecutor - SELLP	<.0001
OclExecutor - CSR vs. OclExecutor - ELL	<.0001
OclExecutor - CSR vs. ReferenceExecutor - ELL	<.0001
OclExecutor - SELLP vs. ReferenceExecutor - SELLP	<.0001
OclExecutor - SELLP vs. OclExecutor - ELL	0.0198
OclExecutor - SELLP vs. ReferenceExecutor - ELL	<.0001
OclExecutor - ELL vs. ReferenceExecutor - ELL	<.0001
ReferenceExecutor - SELLP vs. OclExecutor - ELL	<.0001
ReferenceExecutor - SELLP vs. ReferenceExecutor - ELL	<.0001
ReferenceExecutor - CSR vs. OclExecutor - SELLP	<.0001
ReferenceExecutor - CSR vs. ReferenceExecutor - SELLP	<.0001
ReferenceExecutor - CSR vs. OclExecutor - ELL	<.0001
ReferenceExecutor - CSR vs. ReferenceExecutor - ELL	<.0001

Table 9. Contrasting: Executors, and matrix row size, via p-values (*alpha* = 0.05)

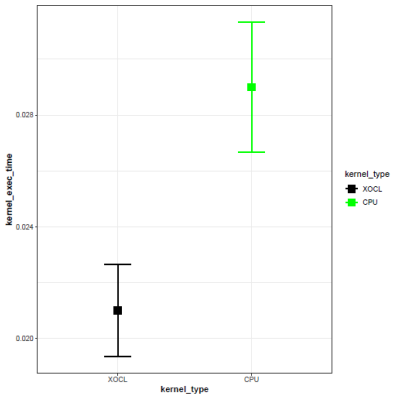
Contrast group	p-value
OclExecutor - 2k vs. ReferenceExecutor - 16k	<.0001
ReferenceExecutor - 2k vs. OclExecutor - 4k	<.0001
ReferenceExecutor - 2k vs. ReferenceExecutor - 4k	<.0001
ReferenceExecutor - 2k vs. OclExecutor - 8k	<.0001
ReferenceExecutor - 2k vs. ReferenceExecutor - 8k	<.0001
ReferenceExecutor - 2k vs. OclExecutor - 16k	<.0001
ReferenceExecutor - 2k vs. ReferenceExecutor - 16k	0.0198
OclExecutor - 4k vs. ReferenceExecutor - 4k	<.0001
OclExecutor - 4k vs. OclExecutor - 8k	<.0001
OclExecutor - 4k vs. ReferenceExecutor - 8k	<.0001
OclExecutor - 4k vs. OclExecutor - 16k	<.0001
OclExecutor - 4k vs. ReferenceExecutor - 16k	<.0001
ReferenceExecutor - 4k vs. OclExecutor - 8k	<.0001
ReferenceExecutor - 4k vs. ReferenceExecutor - 8k	<.0001
ReferenceExecutor - 4k vs. OclExecutor - 16k	<.0001
ReferenceExecutor - 4k vs. ReferenceExecutor - 16k	<.0001
OclExecutor - 8k vs. ReferenceExecutor - 8k	0.0057
OclExecutor - 8k vs. OclExecutor - 16k	<.0001
OclExecutor - 8k vs. ReferenceExecutor - 16k	<.0001
ReferenceExecutor - 8k vs. OclExecutor - 16k	<.0001
ReferenceExecutor - 8k vs. ReferenceExecutor - 16k	<.0001
OclExecutor - 16k vs. ReferenceExecutor - 16k	<.0001
OclExecutor - 2k vs. OclExecutor - 16k	<.0001
OclExecutor - 1k vs. ReferenceExecutor - 1k	0.1546
OclExecutor - 1k vs. OclExecutor – 2k	1.0000
OclExecutor - 1k vs. ReferenceExecutor – 2k	<.0001
OclExecutor - 1k vs. OclExecutor - 4k	1.0000
OclExecutor - 1k vs. ReferenceExecutor - 4k	<.0001
OclExecutor - 1k vs. OclExecutor - 8k	<.0001
OclExecutor - 1k vs. ReferenceExecutor - 8k	<.0001
OclExecutor - 1k vs. OclExecutor - 16k	<.0001
OclExecutor - 1k vs. ReferenceExecutor - 16k	<.0001
ReferenceExecutor - 1k vs. OclExecutor - 2k	0.1356
ReferenceExecutor - 1k vs. ReferenceExecutor - 2k	0.0621
ReferenceExecutor - 1k vs. OclExecutor - 4k	0.1804
ReferenceExecutor - 1k vs. ReferenceExecutor - 4k	<.0001
ReferenceExecutor - 1k vs. OclExecutor - 8k	<.0001
ReferenceExecutor - 1k vs. ReferenceExecutor - 8k	<.0001
ReferenceExecutor - 1k vs. OclExecutor - 16k	<.0001
ReferenceExecutor - 1k vs. ReferenceExecutor - 16k	<.0001
OclExecutor - 2k vs. ReferenceExecutor - 2k	<.0001
OclExecutor - 2k vs. OclExecutor - 4k	1.0000
OclExecutor - 2k vs. ReferenceExecutor - 4k	<.0001
OclExecutor - 2k vs. OclExecutor - 8k	<.0001
OclExecutor - 2k vs. ReferenceExecutor - 8k	<.0001



(a)



(b)



(c)

Figure 5. (a) Relation between the kernel executor type and the matrix number of rows over the kernel execution time response. (b) Relation between the kernel executor type and the SpMV kernel algorithm over the kernel execution time response. (c) Statistical difference between the kernel executor types over the kernel execution time response.

7 Discussion

7.1 Kernels Parallelism

The experiments performed in this work were based on the data parallelism strategy. However, the memory constraints in OpenCL limited the leveraging of the hardware-based acceleration. As indicated before, we got limited to a maximum of 4k HWBs, i.e., if the matrix has more than 4k rows, we will fall into using the chunking algorithm to process blocks of as many of 4k work items. Figure 4 depicts this situation, in which the speedup of all kernels achieves a peak at 4k rows and then begins to lose efficiency. This speedup loss occurs as a result of invoking the OpenCL kernels repeatedly via the chunking approach to process the matrices. The overhead of calling the kernel from the host is identified as the root cause of this problem, which is a performance issue that falls back again on the OpenCL library. Lastly, we can see from Table 6 that employing the maximum number of work items allows us to accelerate the kernel execution time by at least 10x while solving a SpMV computation (the largest speedup happens with the ELL kernel, which can speed the computation by 249x times). These outcomes indicate that our technique is useful, but that hardware optimizations in kernels are required.

7.2 Reference Kernels vs. FPGA-Based

We successfully achieved to demonstrate that reconfigurable hardware-based kernels implemented in Ginkgo generate results that are statistically different than those obtained with the reference kernels, and more importantly, that on average they generate better results in terms of kernel execution time. This can be corroborated with the p-value presented in Table 7, and Figure 5c. Where it is evident how our kernels outperform the results obtained with the reference ones.

The results listed in Table 8, Table 9, and Figure 5 help to granularly see the effects that every component under analysis has in the kernel execution time (like the number of rows effect, which help to drive to similar conclusions made above). An important detail from this analysis is that the reference CSR kernel outperforms ours, this happens because of the nature of the algorithm. Regarding the state of the art, CSR SpMV kernels are better suited to work on CPUs, due to the high caching they provide (Anzt, Tomov, & Dongarra, 2014). In systems with low cache hierarchy like FPGAs, the parallelism is compromised due to the high memory requirements. The second point that generates this low performance is also related to memory, but the root cause is the *nnz* in the matrix because the *row_ptr* grows about it, which implies more memory is needed.

8 Concluding Remarks

FPGAs have a promising future in HPC, boosted with the usage of HLS like OpenCL one can generate compute units for linear algebra computations. In our approach, we generated a kernel-specific based solution, which accelerates all the algorithm body of the CSR, SELLP, and SELL SpMV kernels formats, and obtained speedups of at least 10x when using 4k HWBs. Furthermore, we demonstrated via a performance characterization study that FPGAs outperform general-purpose processors in terms of compute time. Finally, we achieved our overall objective of extending the Ginkgo library to use reconfigurable hardware-based kernels.

In regard to future work, we intend to extend this study by implementing hardware optimizations to the kernels generated and designing compute units to accelerate specific code blocks in kernels and not the whole kernel code as we did in this approach. We also plan to replace OpenCL with XRT Native (AMD, 2022b) since it is more optimized for Xilinx devices and allows direct integration with C++. Additionally, we think that implementing arrays of FPGAs is another promising idea because one can leverage the power of more than one device at the same time.

Acknowledgements

This work was supported in part by AMD under the Heterogeneous Accelerated Compute Cluster (HACC) program.

ORCID ID

Alejandro Morales-Peña https://orcid.org/0009-0004-9508-4266
Esteban Meneses https://orcid.org/0000-0002-4307-6000

References

AMD. (2022a, August 4). *Heterogeneous Accelerated Compute Cluster (HACC) Program*. (Advanced Micro Devices, Inc) Retrieved 2023, from AMD Website: https://www.amd.com/en/corporate/university-program/aup-hacc.html

AMD. (2022b, October 7). *XRT Native APIs*. (Advanced Micro Devices, Inc) Retrieved 2023, from https://xilinx.github.io/XRT/master/html/xrt_native_apis.html

AMD. (2023). *ROCm Software 5.3.0: HIP Documentation*. (Advanced Micro Devices, Inc) Retrieved 2024, from AMD website: https://rocm.docs.amd.com/projects/HIP/en/docs-5.3.0/index.html

AMD. (2024, May 15). *AMD*. (Advanced Micro Devices, Inc) Retrieved 2024, from AMD Website: https://www.amd.com/en.html

Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J., . . . Sorensen, D. (1999). *LAPACK Users' Guide* (Third ed.). Philadelphia, USA: SIAM. doi:10.1137/1.9780898719604

Anzt, H., Cojean, T., Chen, Y.-C., Flegar, G., Göbel, F., Grützmacher, T., . . . Tsai, Y.-H. (2020). Ginkgo: A high performance numerical linear algebra library. *Journal of Open Source Software, 5*(52), 1-6, 2260. doi:10.21105/joss.02260

Anzt, H., Cojean, T., Flegar, G., Göbel, F., Grützmacher, T., Nayak, P., . . . Quintana-Ortí, E. S. (2022, March). Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing. (Z. Bai, & W. Bangerth, Eds.) *ACM Transactions on Mathematical Software (TOMS), 48*(1), 1-33, Article No. 2. doi:10.1145/3480935

Anzt, H., Tomov, S., & Dongarra, J. (2014, April). *Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-formats on NVIDIA GPUs.* Technical Report UT-EECS-14-727, University of Tennessee. Retrieved from https://icl.utk.edu/files/publications/2014/icl-utk-772-2014.pdf

Bosch, J., Tan, X., Filgueras, A., Vidal, M., Mateu, M., Jiménez-González, D., . . . Labarta, J. (2018). Application Acceleration on FPGAs with OmpSs@FPGA. In *2018 International Conference on Field-Programmable Technology (FPT), Naha, Japan, 10-14 Dec.* (pp. 70-77). IEEE. doi:10.1109/FPT.2018.00021

BSC. (2016). *Linear Algebra and Math Libraries*. (Barcelona Supercomputing Center) Retrieved 2023, from BSC website: https://www.bsc.es/research-development/research-areas/programming-models/linear-algebra-and-math-libraries

Cppreference. (2024, October 4). *RAII*. Retrieved from Cppreference website: https://en.cppreference.com/w/cpp/language/raii

Davis, T. A., & Hu, Y. (2011, November). The university of Florida sparse matrix collection. *ACM Transactions on Mathematical Software, 38*(1), 1-25, Article 1. doi:10.1145/2049662.2049663

De Matteis, T., de Fine Licht, J., & Hoefler, T. (2020). fBLAS: streaming linear algebra on FPGA. *SC '20: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, November 9 - 19* (pp. 1-13, Article 59). Atlanta, Georgia, USA: IEEE. doi:10.5555/3433701.3433779

Dongarra, J. J., & Walker, D. W. (1995). Software Libraries for Linear Algebra Computations on High Performance Computers. *SIAM Review, 37*(2), 151-180. doi:10.1137/1037042

Dongarra, J., & Blackford, L. S. (1996). ScaLAPACK tutorial. In J. Waśniewski, J. Dongarra, K. Madsen, & D. Olesen, *Applied Parallel Computing Industrial Computation and Optimization. Third International Workshop, PARA 1996, Lyngby, Denmark, August 18-21. Lecture Notes in Computer Science* (Vol. 1184, pp. 204–215). Berlin, Heidelberg, Germany: Springer. doi:10.1007/3-540-62095-8_22

ETH Zürich. (2024). *ETH Zürich*. Retrieved from ETH website: https://ethz.ch/de.html

Fang, J., Mulder, Y. B., Hidders, J., Lee, J., & Hofstee, H. P. (2020, January). In-memory database acceleration on FPGAs: a survey. *The VLDB Journal, 29*(1), 33–59. doi:10.1007/s00778-019-00581-w

Gao, Y., & Zhang, P. (2016). A Survey of Homogeneous and Heterogeneous System Architectures in High Performance Computing. *2016 IEEE International Conference on Smart Cloud (SmartCloud), 8-20 Nov.* (pp. 170-175). New York, NY, USA: IEEE. doi:10.1109/SmartCloud.2016.36

Girden, E. R. (1992). *ANOVA: repeated measures.* Newbury Park, CA, USA: Sage, University Paper Serires on Quantitativer Aplications in the Social Sciences, Series 07-084. doi:10.4135/9781412983419

Gonzalez, J., & Núñez, R. C. (2009, July 1). LAPACKrc: Fast linear algebra kernels/solvers for FPGA accelerators. *Journal of Physics: Conference Series, SciDAC 2009, 14–18 June, 180*(1, 012042). doi:10.1088/1742-6596/180/1/012042

Kestur, S., Davis, J. D., & Chung, E. S. (2012). Towards a Universal FPGA Matrix-Vector Multiplication Architecture. *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, 29 April - 1 May* (pp. 9-16). Toronto, ON, Canada: IEEE. doi:10.1109/FCCM.2012.12

Khronos Group. (2024). *OPenCL: Open Standard for Parallel Programming of Heterogeneous Systems*. Retrieved from Khronos Group website: https://www.khronos.org/opencl/

Khronos Group. (2024). *SYCL: C++ Programming for Heterogeneous Parallel Computing*. Retrieved from Khronos® Group website: https://www.khronos.org/api/index_2017/sycl

Kuon, I., Tessier, R., & Rose, J. (2008). FPGA Architecture: Survey and Challenges. *Foundations and Trends in Electronic Design Automation, 2*(2), 135-253. doi:10.1561/1000000005

Lawson, C. L., Hanson, R. J., Kincaid, D. R., & Krogh, F. T. (1979, September). Basic Linear Algebra Subprograms for Fortran Usage. (J. R. Rice, Ed.) *ACM Transactions on Mathematical Software (TOMS), 5*(3), 308–323. doi:10.1145/355841.355847

NVIDIA Corporation. (2024). *CUDA Toolkit*. Retrieved from NVIDIA Developer website: https://developer.nvidia.com/cuda-toolkit

OpenMP. (2024). *OpenMP: The OpenMP API specification for parallel programming*. Retrieved from OpenMP website: https://www.openmp.org/

Podobas, A. (2014). Accelerating Parallel Computations with OpenMP-Driven System-on-Chip Generation for FPGAs. *Proceedings of the 2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs, MCSOC '14, September 23 - 25* (pp. 149-156). Washington, DC, USA: IEEE. doi:10.1109/MCSoC.2014.30

Sommer, L., Korinth, J., & Koch, A. (2017). OpenMP device offloading to FPGA accelerators. *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP), 10-12 July* (pp. 201-205). Seattle, WA, USA: IEEE. doi:10.1109/ASAP.2017.7995280

Steffenel, L. A. (2019). HPC challenges for the next years: the rising of heterogeneity and its impact on simulations. *CECAM Workshop: Microscopic simulations: forecasting the next two decades, April 24-26* (pp. 1-25). Toulouse, France: CECAM - Centre Européen de Calcul Atomique et Moléculaire. Retrieved from https://hal.univ-reims.fr/hal-02120029

Sun, J., Peterson, G. D., & Storaasli, O. (2007). Mapping Sparse Matrix-Vector Multiplication on FPGAs. *Proceedings of the Third Annual Reconfigurable Systems Summer Institute (RSSI'07), July 17-20* (pp. 1-10). Urbana, Illinois, USA: RSSI. Retrieved from http://rssi.ncsa.illinois.edu/2007/proceedings/papers/rssi07_12_paper.pdf

Townsend, K. R. (2016). *Computing SpMV on FPGAs.* PhD Thesis, Iowa State University, Electrical and Computer Engineering, Ames, Iowa. doi:10.31274/etd-180810-4826

Tsoi, K. H., & Luk, W. (2010). Axel: a heterogeneous cluster with FPGAs and GPUs. *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays,*

FPGA '10, Monterey, California, USA, February 21 - 23 (pp. 115–124). New York, NY, USA: Association for Computing Machinery. doi:10.1145/1723112.1723134

Zhang, Z., Fan, Y., Jiang, W., Han, G., Yang, C., & Jason, C. (2008). AutoPilot: A Platform-Based ESL Synthesis System. In P. Coussy, & A. Morawiec (Eds.), *High-Level Synthesis: From Algorithm to Digital Circuit* (pp. 99-112). Dordrecht, Netherlands: Springer. doi:10.1007/978-1-4020-8588-8_6

Zhuo, L., & Prasanna, V. K. (2005). High Performance Linear Algebra Operations on Reconfigurable Systems. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, 12-18 November* (p. 2). Seattle, WA, USA: IEEE. doi:10.1109/SC.2005.31