Author's copy (preprint) of Paolo Viviani, Maurizio Drocco, Marco Aldinucci, and Marco Grangetto. Accelerating spectral graph analysis through wavefronts of linear algebra operations. In Proc. of 27th Euromicro Intl. Conference on Parallel Distributed and network-based Processing (PDP), pages 9–16, Pavia, Italy, 2019. IEEE. doi: 10.1109/ EMPDP.2019.8671640.

Accelerating spectral graph analysis through wavefronts of linear algebra operations

Maurizio Drocco
Pacific Northwest National Laboratory
Richland, WA, USA
maurizio.drocco@pnnl.gov

Paolo Viviani
Noesis Solutions NV, Belgium
Computer Science Department
University of Torino, Italy
paolo.viviani@noesissolutions.com

Iacopo Colonnelli
Marco Aldinucci
Marco Grangetto
Computer Science Department
University of Torino, Italy
iacopo.colonnelli@unito.it
{aldinuc, grangetto}@di.unito.it

***Abstract*—The wavefront pattern captures the unfolding of a parallel computation in which data elements are laid out as a logical multidimensional grid and the dependency graph favours a diagonal sweep across the grid. In the emerging area of spectral graph analysis, the computing often consists in a wavefront running over a tiled matrix, involving expensive linear algebra kernels. While these applications might benefit from parallel heterogeneous platforms (multi-core with GPUs), programming wavefront applications directly with high-performance linear algebra libraries yields code that is complex to write and optimize for the specific application. We advocate a methodology based on two abstractions (linear algebra and parallel pattern-based run-time), that allows to develop portable, self-configuring, and easy-to-profile code on hybrid platforms.**

***Index Terms*—linear algebra,GPU,wavefront,hybrid**

I. Introduction

Graph signal processing is an emerging technique that can be applied in a broad range of topics as, for instance, image compression, epidemiological data, logistics and many other fields [1]. In this paper we present a novel approach aimed to address a number of computational challenges related to the application of graph signals to image processing.

In order to identify and study such challenges without restricting ourselves to a specific processing algorithm, we propose a representative mock-up of a block-based image processing procedure that can be mapped to a broad class of similar problems, synthesized in Fig. 1: given an input matrix A of size $n \times m$ that represents the original image bound to be processed, it is logically partitioned into fixed-size squared sub-matrices that will be hereafter referred to as *tiles* or blocks. On each tile is applied a function (*kernel*) that implements some linear algebra processing and returns the transformed block. Depending on the details of the specific algorithm, a pattern of data dependencies can possibly be defined on the two-dimensional grid of tiles—which prevents the developer to speed up the computation by implementing a naive data parallel approach—and should be addressed with specific techniques in order to exploit parallelism by means of so-called *wavefront* patterns [2].

A fundamental parameter that characterises the depicted class of algorithms is the tile size. It depends on the specific application, but it is usually constrained by the computational

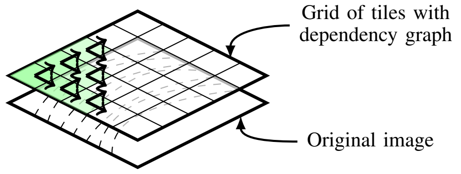


Fig. 1. Representation of the use-case

cost: in fact the complexity of the linear algebra kernel usually as $O(n^3)$. Parallel and heterogeneous platforms (CPU+GPU), may be exploited to achieve high-performance, but a huge effort in terms of coding and optimization is generally required to deal with such platforms.

Starting from the depicted scenario, this paper proposes a general methodology to address the low-effort development of a broad class of applications and their deployment on heterogeneous architectures. The contributions presented here can be resumed in three main points:

1) an approach that integrates a number of techniques and tools to accelerate wavefronts of linear algebra kernels on heterogeneous architectures with very limited effort from the developer;
2) a methodology based on the approach at point 1) that allows to thoroughly characterise the performance of heterogeneous platforms by investigating a large number of parameters of both the problem itself, e.g. tile size, and of the software, e.g. CPU/GPU work balance;
3) an efficient implementation of a graph signal processing use-case.

The paper proceeds as follows. Sec. II presents the details of the application that is considered as both use case and benchmark for this work. Sec. III reviews the most significant related work. Sec. IV describes the software components and the methodologies exploited in order to achieve the goal of this paper. Sec. V reports the results of the benchmarks employed to validate the presented approach. Finally, Sec. VI concludes and outlines some possible future works.

II. Running Example

Signal processing based on graphs has recently emerged as a promising approach in many areas of signal analysis and representation [3]. This field of research builds around the idea of modelling correlations among signal samples as undirected graphs defined as $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, \mathbf{W}\}$, with vertexes $\mathcal{V}$ representing the signal samples, edges $\mathcal{E}$ representing the connections between pairs of correlated samples, and $\mathbf{W}$ a symmetric $n \times n$ weighted adjacency matrix, where n is the number of signal samples. The above graph-based representation allows to exploit spectral graph theory [4] to extend the root operator of signal processing history: the Fourier transform. Namely, the *Generalised Fourier Transform* (GFT) can be defined, that is the spectral representation obtained using as basis functions the eigenvectors of the graph Laplacian.

Depending on the analysis goals, several definitions of the Laplacian matrix can be used, a popular one being the combinatorial Laplacian $\mathbf{L} = \mathbf{D} - \mathbf{W}$, where $\mathbf{D}$ is the degree matrix (i.e., the diagonal matrix which i-th diagonal element d_i is equal to the sum of the weights of all edges incident to node i). GFT is based on the eigen decomposition of $\mathbf{L}$, defined as follows, where $\mathbf{L}$ is a symmetric positive semi-definitive matrix, $\mathbf{U}$ is the matrix whose rows are the eigenvectors of $\mathbf{L}$, and $\mathbf{\Lambda}$ is the diagonal matrix whose diagonal elements are the corresponding eigenvalues.

$$\mathbf{L} = \mathbf{U}^T \mathbf{\Lambda} \mathbf{U} \tag{1}$$

Given a vector $\mathbf{x}$ of samples, the GFT transform and its inverse are defined as:

$$\mathbf{y} = \mathbf{U}\mathbf{x} \qquad \mathbf{x} = \mathbf{U}^T\mathbf{y} \tag{2}$$

Analogously to the Fourier representation, small eigenvalues corresponds to low frequencies and large ones correspond to high frequencies. Many applications leverage on GFT properties, ranging from interpolation of structured data [5] to image compression [6]–[8]. The key to the success of GFT is its ability to match the signal local structure and correlation, that in turns enhances performance in the application domain, e.g. image compression efficiency.

Unfortunately, the GFT advantages are paid in terms of computational cost: given an input signal (e.g., a bitmap image), an eigen decomposition needs to be computed for each *tile* (e.g., a 2D block) in the signal. Since the GFT builds and processes an adjacency matrix for each tile, the actual size of each eigenvector problem is $t^2 \times t^2$, where t is the size of the tile.

In this work, we present a generic mock-up that performs tile-based processing over a 2D signal (e.g., a bitmap image). The mock-up is parametric with respect to the GFT calculation that is performed on each tile of the input image. We will refer hereafter to such calculation as the *GFT kernel*. Since this work is not focused on the details of the image processing algorithms, the GFT kernel implemented here involves only the calculation of terms in Eq. 1 and 2, where the matrix L is generated from the tile in an arbitrary way. Also the dependencies between different tiles are introduced in an arbitrary way, in order to evaluate their impact on the performance.

III. Related Work

From a high-level perspective, this work addresses the need to achieve a reasonable speedup on a large class of problems while keeping the development effort to a minimum. This goal is pursued by combining a set of existing tools, recapped in Sec. IV-A1, in order to handle the two aspects of the computation: the *parallel coordination* supporting task parallelism and the *linear algebra* involved in the GFT kernel. For the parallel coordination, we rely on FastFlow [9], [10], but several tools could have been exploited, like Intel TBB or OpenMP, that also provide interfaces to implement task-parallel programs [11]. For linear algebra functions, we rely on *Dynamic-Armadillo* [12], a customised version of Armadillo[1] (cf. IV-A1), but alternative approaches could rely on, for instance, ArrayFire[2], which would equally meet the requirements for this work; in alternative, a low-level approach could be considered, as, for instance, direct interacting with Magma [13].

A symmetrical approach consists in working at a lower level, that enables a whole different set of techniques to speed up the execution of such workloads on GPUs [14]. However, this increases the development effort and, in most cases, restricts the implementation to a specific application. In our specific case, in order to benefit from a number of optimizations (i.e. efficient host-device memory transfer strategies) that would push the GPU performance slightly further than what achieved with the presented approach, it would be required an unaffordable development effort.

IV. Approach

In this section, we propose a methodology for accelerating wavefronts of linear algebra kernels on heterogeneous platforms. We formulate the discussion around the example introduced in Sec. II, but a generalisation to different linear algebra kernels would be straightforward. We rely on the Armadillo and FastFlow tools, recapped in Sec. IV-A1. We proceed by decomposing the problem into two simpler subproblems. In Sec. IV-B, various heterogeneous deployments (i.e., multicore CPUs with GPU accelerators and SOCs) are considered over a simplified GFT kernel, in which inter-tiles dependencies are dropped. In Sec. IV-C, the simplified kernel is endowed with the logic for managing a wavefront dependency pattern.

A. Tools

1) Armadillo: Armadillo is a C++ template library for linear algebra with an high-level API, which is deliberately similar to Matlab and provides functions to manipulate custom objects representing vectors, matrices and tensors; it relies on an underlying BLAS and LAPACK implementation (hereafter

[1]http://arma.sourceforge.net/
[2]https://arrayfire.com

we will refer to such implementation as the back-end) to achieve high performance. The textbook-like API significantly lowers the barrier to high performance numerical linear algebra. For instance, the calculation of eigenvectors, required by GFT, in LAPACK has a monstrous 11-parameter syntax, whereas in Armadillo it is a simple 3-parameter function, to be invoked as: `eig_sym(e, E, A)`.

In this work, we exploit Dynamic-Armadillo [12], an extension that supports dynamic offloading of operations onto multiple back-ends. Specifically, the OpenBLAS [15] CPU back-end and the Magma [13] hybrid CPU/GPU back-end will be used. In this context, this set-up makes it possible to investigate how GPU usage within Magma affects the overall performance and to model under which conditions the usage of a hybrid CPU-GPU back-end is beneficial.

Notice that, by employing Armadillo, we can only execute each linear algebra operation on the tiles independently, while the parallelism is exploited at a higher level. This approach is not ideal from a performance perspective if compared to a purely batched approach, like Magma batched routines [16], [17]. Nevertheless, two issues prevented us from exploiting batched routines: 1) batched `ssyevd` is not currently provided in Magma (v2.2); 2) it would be required a much more complex dependency management and a deeper knowledge of the complex LAPACK/Magma API, conflicting with the low-effort approach we advocate.

2) FastFlow: FastFlow [10] is a C++ header-only library for parallel programming. Among the different levels targeted by the FastFlow API, in this work we consider the core patterns interface, where programs are arbitrary compositions of *farm* and *pipeline* patterns. In particular, we only use the farm pattern.

In a FastFlow farm, an emitter is connected to n workers, each attached to the farm collector. Arbitrary policies can be implemented for distributing tokens to the workers. Since the emitter acts as a centralization point, it is sufficient to embed the policy into the emitter routine as sequential C++ code. With other approaches, based for instances on concurrent data structures (e.g., a shared work pool), it would be required to encode the policy as a distributed protocol across the concurrent executors. We exploit such feature in Sec. IV-C for encoding the management of wavefront dependencies.

Although FastFlow has been proven to be as fast as similar frameworks such as Intel TBB or OpenMP [18], we consider the pure performance of the coordination mechanisms out of the scope of this work. The performance on the presented benchmark depends mostly on the implementation of the linear algebra kernel.

B. Multi-Platform Tiled Matrix Processing

We consider the simplified problem of tiled matrix processing. Given an input matrix A, it is logically partitioned into fixed-size squared tiles and each tile is processed, independently from each other, by a function f, that we call *kernel*. We denote by $\tilde{a}_{i,j}$ the input tile at position (i, j) in the two-dimensional tile space and we define the output matrix B such that $\tilde{b}_{i,j} = f(\tilde{a}_{i,j})$ for each tile.

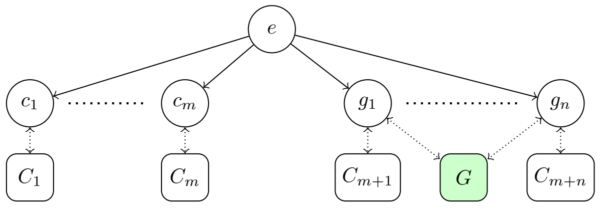


Fig. 2. general deployment setting; c_i and g_i represent the i-th CPU worker and GPU worker, respectively, according the FastFlow numeration of farm workers; C_i represents the i-th CPU context according to the OS numeration; G represents the GPU.

In this work, the kernel function is a representative GFT procedure (Sec. II), that we implemented in C++ on top of the Armadillo library. As recapped in Sec. IV-A1, Armadillo provides two variants for each function, namely CPU-only and mixed CPU+GPU, yielding two variants of the GFT procedure. Note that, for brevity, we will refer hereafter to the Magma back-end as the GPU back-end.

1) Data-Parallel Setting: In Fig. 2 is depicted the general setting for a data-parallel implementation of the depicted problem, that consists in a FastFlow farm composed of m CPU workers and n GPU workers. A CPU worker executes the CPU-only GFT and it is attached to a CPU context, whereas a GPU worker executes the mixed GFT and it is attached to both a CPU context and the GPU.

We categorise the deployments into two classes, namely *homogeneous* and *heterogeneous*. In a deployment of the former class all the workers are either CPU or GPU workers (i.e., $n = 0$ or $m = 0$), whereas in a deployment of the latter class they can be mixed. The workers are fed by an emitter node e, that performs the matrix partitioning and distributes the input tiles to the workers with either a *round-robin* policy or an *on-demand* policy that sends each tile to the first inactive worker, for better load balancing..

2) Deployments: We consider two deployments for exploiting a multicore CPU, without taking into account any additional accelerator (i.e., homogeneous CPU deployments with $n = 0$). In particular we consider two different values for m: either equal to the number of physical core or to the number of CPU contexts (e.g., two per physical core in a typical hyper-threaded platform). The former case is expected to show a neat scalability up to m, as long as the working set for each input tile fits in some level of cache. The latter deployment should instead not provide any substantial gain from multi-context processing, since the working sets for each input tile are disjoint (i.e. no cooperative caching).

For a single GPU worker mapped to a single CPU core and the GPU ($n = 1$), we observe a performance gain if and only if the average time needed for processing a tile by the GPU worker is smaller than the average time needed by a CPU worker. Although with this accelerator offloading we increase the amount of available parallelism—in particular in case of highly parallel architectures such as GPUs—the above condition is not always verified because of non-negligible overheads induced by the underlying platform model.

Furthermore, considering a configuration in which the above condition holds, although it sounds natural trying to increase n—if a single GPU worker performs better than its CPU counterpart, one expects to observe a performance gain by replacing n CPU workers with n GPU ones—the GPU is a shared resource, exploited concurrently by all the GPU workers, thus limiting the scalability. Therefore, we consider deployments in which $n > 0$ GPU workers are coupled with $m > 0$ CPU workers in an heterogeneous farm.

C. *Wavefront Processing*

In Sec. IV-B we considered blocked matrix processing, in which each output tile $\tilde{b}_{i,j}$ is function only of the corresponding input tile $\tilde{a}_{i,j}$, namely $\tilde{b}_{i,j} = f\left(\tilde{a}_{i,j}\right)$. As outlined in Sec. II, some GFT methods introduce a more complex dependency relation: for instance, we consider the common case with $\tilde{b}_{i,j} = f\left(\tilde{a}_{i,j}, \tilde{b}_{i-1,j}, \tilde{b}_{i,j-1}\right)$. This definition breaks the simple data-parallel nature and introduces a dependency pattern—often referred as the *wavefront* pattern—in which the computation for the output tile $\tilde{b}_{i,j}$ cannot start until the computations for tiles $\tilde{b}_{i-1,j}$ and $\tilde{b}_{i,j-1}$ have been completed.

We implemented the depicted wavefront dependency pattern as an instance of the FastFlow construct *farm-with-feedback*, that adds a feedback channel from each worker back to the emitter node (cf. Fig. 2). When a tile $\tilde{b}_{i-1,j}$ is completed, (a reference to) it is sent back to the emitter that updates its internal state and possibly schedules for execution the tiles $\tilde{b}_{i+1,j}$ and $\tilde{b}_{i,j+1}$.

V. Experimental Validation

In this section we provide some experimental evidence of the conjectures about performance we formulated in the previous sections.

We used two platforms for the reported experiments:

A) dual-socket Intel Xeon E5-2680 v3 running at 2.50GHz with 24 cores (12 per socket), equipped with two Nvidia Tesla K40 GPUs;
B) Nvidia Jetson TX1 equipped with a 4-core ARM A7 CPU and an embedded Nvidia Maxwell GPU.

Each experiment has been repeated 4 times and the median values are reported. Since all the measurements exhibited negligible variance, we omitted error quantification.

A. *Eigenvector computation*

The `ssyevd` routine (i.e., Symmetric Eigenvectors) dominates the GFT execution time, therefore we isolate the study of its performance in order to better understand the global performance of the application.

Fig. 3a compares the performance of `ssyevd` in its CPU and GPU variants (respectively OpenBLAS and Magma in the figure), with respect to the square size of the input matrix. From the reported results, we infer that Magma forces using the CPU variant for small (i.e., with square size up to 128) input matrices. The GPU variant performs worse than the CPU one until the square size reaches 2048, where the execution times of the two variants collapse. Finally, for bigger input matrices the GPU variant performs better.

From the above considerations, we expect to observe some performance gain from the GPU only for sufficient tile sizes (i.e. from $\sqrt{2048} \simeq 45$). It is noteworthy that Fig. 3b show exactly the same trend as figure Fig. 3a, while actually running on a completely different architecture. This can be surprising given that the CPU and the GPU share the same memory and no transfer is paid when offloading a task to the GPU, potentially improving the performance on small problem sizes. Nevertheless, the Jetson TX1 graphic processor presents a much slower (25.6 GB/s vs 288 GB/s) memory bandwidth and a narrower (64-bit vs. 384-bit) memory bus that prevent the GPU to fully benefit from the lack of memory transfer. In this context we can indeed consider the TX1 as a scaled-down Xeon node, where the balance of processing power between the CPU and the GPU is, for the specific workload, roughly the same.

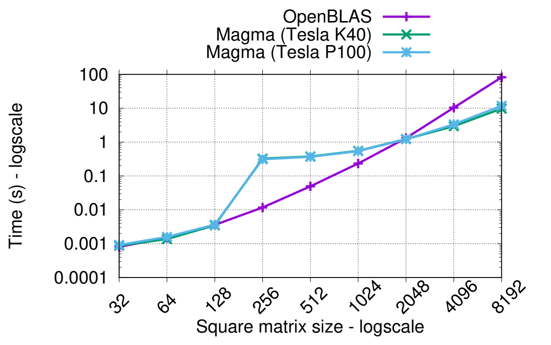
B. *Data-Parallel Tiled Matrix Processing*

1) Multicore Deployments: In Sect. IV-B2 we stated that, under certain conditions, we expect linear scalability with respect to the number of CPU cores in a homogeneous multicore deployment. The performance results in Figures 4a, 4b and 4c confirm the hypothesis above for tile sizes 8, 16, and, to a slightly lesser degree, 32 respectively. Conversely, the speedup is strongly sub-linear for tile size 64. The working set in such case is large at least as the adjacency matrix computed by GFT—namely $64^4 = 16.8M$ items, that is 67 MB with 32-bit floating point items—thus even the most external level of cache gets filled rapidly and the scalability is bound by the CPU-memory bandwidth.

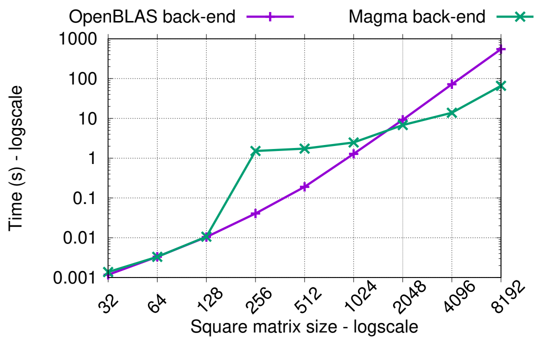
Also the behaviour with respect to multiple CPU contexts confirms the hypotheses we formulated, thus we observe a very limited performance gain if we deploy more CPU workers than physical CPU cores. In such case it is important to schedule tiles on demand (rather than round-robin) since some workers are somehow penalised by the OS thread scheduling. It is indeed possible to notice from the performance drop of the rightmost point of Fig. 4a and 4e that, for tile size equal to 8, the on-demand scheduler heavily loads the core where the emitter is allocated, actually reducing the number of available cores by one.

2) GPU Deployments: The basic setting to understand the performance of homogeneous GPU deployments as presented in Sec. IV-B can be found in Figures 4 for each tile size at the points with abscissa 1 on the GPU lines.

As we estimated in Sec. V-A, the case in Fig. 4b with tile size 16 (that corresponds to input size 256 for `ssyevd`) is a worst-case scenario for GPU deployments. The case in Fig. 4c with tile size 32 (1024 for `ssyevd`) performs better, but still slower than the CPU deployment, whereas the case in Fig. 4d with tile size 64 performs better than the CPU. In all the cases, we observe poor scalability when adding GPU



(a) Xeon E5-2680 v3 @ 2.50GHz + Nvidia Tesla K40 + Nvidia Tesla P100.



(b) SOC with 4 64-bit ARM A57 CPUs + 256-core CUDA Maxwell graphic processor.

Fig. 3. Computation of symmetric eigenvalues using `ssyevd`, comparison between OpenBLAS and Magma back-ends for different problem sizes.

workers: this is due to the GPU being a shared resource. The behaviour exposed by platform B is the same, as also expected from Sec. V-A. The poor scalability for tile size 64 also highlights a further issue: longer GPU bursts potentially leave the CPU idle while the GPU works, inducing CPU under-utilization that plays an important role in exploiting heterogeneous deployments, as we discuss in the next section.

In Fig. 5 we present a simple performance model that reflects the results of Sec. V-A.

If we compare the performance reported in Sec. V-A to the presented model, it is possible to argue that, by raising the tile size, the interactions between the CPU and GPU moves from the situation in phase A to the situation in phase B. Moreover, with larger and larger input sizes we would observe a new condition (phase C in the figure), in which the computation is performed mainly by the GPU. Such scenario takes advantage of the high degree of GPU parallelism, resulting in a reduction of the execution time. However, it both induces CPU under-utilization and occupies GPU resources, thus reducing the scalability in a configuration with multiple GPU workers, as seen in Fig. 4d. To overcome this issue, it is possible to introduce more CPU-only workers, with the benefit discussed in the next section.

This model can help to understand the peculiar implementation of Magma: in fact it employs a task-based approach that schedules a complex DAG of tasks either on the CPU or on the GPU based on the specific workload. Operations exhibiting true data dependencies among data, and global synchronizations are typically executed on the CPU. When calling a magma function, depending on the specific operation and data size, only a fraction of the computation is actually offloaded to the GPU. The fraction of the kernel performed on the CPU is indeed not known in advance, but it is definitely not negligible.

3) Heterogeneous Deployments: Considering what discussed above, it is expected that employing both the CPU and the GPU together would improve the overall performance due to: i) additional CPU workers that overcome the limited scalability showed by the GPU alone and ii) possibility to avoid the CPU underutilization by filling it with additional CPU workers.

In order to quantify the impact of effect i), we performed the test presented in Fig. 6 with mixed CPU and GPU workers: in particular it is possible to identify exactly where the number of GPU workers starts to saturate the GPU queue and the subsequent latency starts to impact the performance. The resulting plot shows how adding more than 12 GPU workers negatively affects the performance.

Effect ii) has been taken into account by employing additional CPU workers with respect to the optimal point of Fig. 6. The best result has been recorded with 20 CPU workers and 12 GPU workers, with a speedup of 18.86.

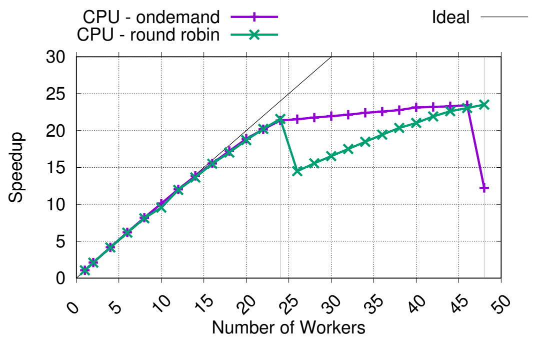
Also the performance measured for heterogeneous deployments can be explained by the model in Fig. 5. In particular., the model depicts the relative time spent by Magma running on the CPU and the GPU with respect to a CPU-only approach performing the same operation.

The performance analysis carried out on heterogeneous architectures represents, in the authors' view, the most interesting result of this work beside the application itself. In fact, the complex behaviour of a black-box, third-party code is inferred from a simple benchmarking set-up.

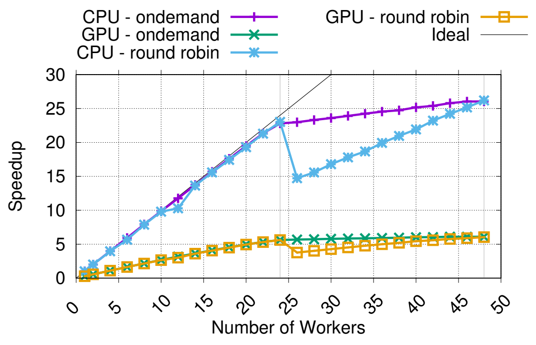
C. Wavefront

In order to evaluate the impact of wavefront dependencies (cf. Sec. IV-C), we consider the deployments leading the best execution times for the data-parallel GFT variant, for each tile size. As reported in Table I, the impact of dependencies grows with the tile size.

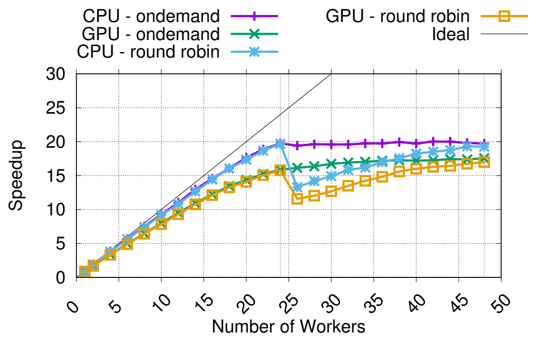
In general, wavefront patterns induce some *transient* phases in which only a limited set of tiles are ready to be processed. Within a transient phase, the available parallelism is limited and the execution exhibits poor scalability, therefore the impact of the dependencies depends on how much time is spent in such transient phases. The pattern we consider in this work (i.e., top and left dependencies) induces such behaviour only during initial and final phases. Since only a fraction of the processed tiles are processed within such phases, the impact



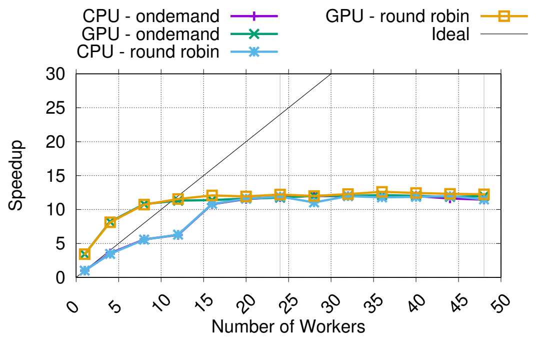
(a) OpenBLAS only, tile size = 8.



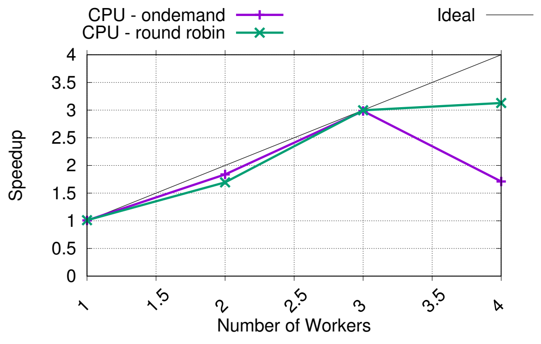
(b) OpenBLAS and Magma, tile size = 16.



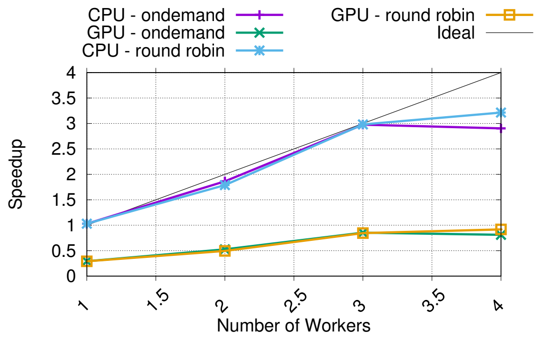
(c) OpenBLAS and Magma, tile size = 32.



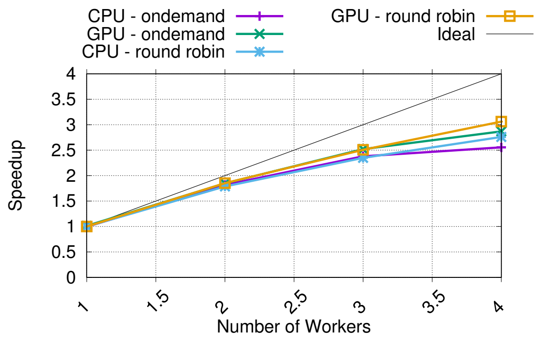
(d) OpenBLAS and Magma, tile size = 64. As expected from Fig. 3a, the GPU provides super-linear speedup for up to 8 workers.



(e) OpenBLAS only, tile size = 8.



(f) OpenBLAS and Magma, tile size = 16.



(g) OpenBLAS and Magma, tile size = 32.

Fig. 4. Speedup for purely data-parallel version. Figs. 4a to 4d, 6 are run on 2× 12 core Xeon E5-2680 v3 @ 2.50GHz + Nvidia Tesla K40, Figs. 4e to 4g are run on SOC with 4 64-bit ARM A57 CPUs + 256-core CUDA Maxwell. GPU deployment not shown in Fig. 4a, 4e since Magma reverts to the CPU for small matrices.

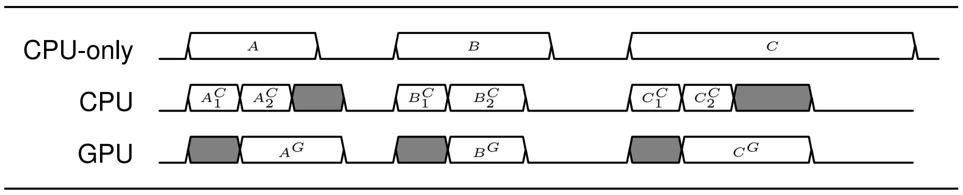


Fig. 5. Schematic representation of a Magma routine execution with GPU offloading. Three different load balancing scenarios that can be related to different problem sizes.

TABLE I
TIMING AND SPEEDUP FIGURES FOR OPTIMAL WAVEFRONT DEPLOYMENT. ORIGINAL IMAGE SIZE: 1024×1024. $2\times$ 12 CORE XEON E5-2680 V3 @ 2.50GHZ + NVIDIA TESLA K40.

Tile size	Sequential time	Best time	Speedup	Optimal deployment
8	14.99	0.75	19.99	24 CPU workers, round-robin
16	50.1	2.28	21.97	24 CPU workers, round-robin
32	255.7	17.70	14.44	24 CPU workers, round-robin
64	2550.4	304.7	8.37	15 CPU + 9 GPU workers, on-demand

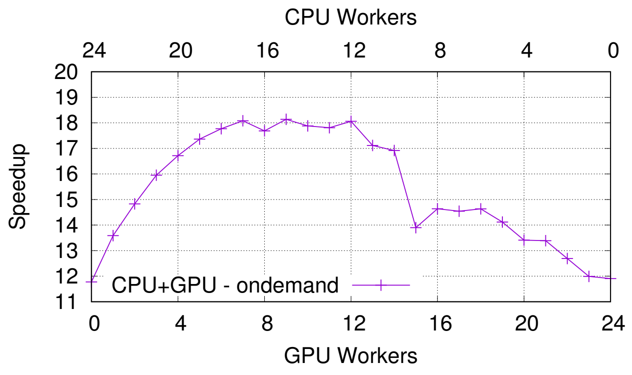


Fig. 6. Mixed CPU/GPU deployment: 24 total workers, variable CPU/GPU share from $(24, 0)$ to $(0, 24)$. 12 core Xeon E5-2680 v3 @ 2.50GHz + Nvidia Tesla K40.

of dependencies and the total number of processed tiles result to be inversely proportional.

The above considerations are confirmed by comparing the experimental results in Table I with the performance of the data-parallel variant in Fig. 4. For small tiles (i.e., size 8 and 16), the maximum speedup is not far from the data-parallel case, whereas for larger tiles (i.e., size 32 and 64) the transient phases become non negligible. It should be noted that, for a tile size of 64, a sample image of size 1024×1024 actually limits the maximum wavefront size to 16 (i.e., the number of tiles laying on the longest matrix diagonal), that represents the total scalable parallelism available and hence lays an upper bound to the speedup.

VI. CONCLUSION

In this work, we proposed an approach for developing, analysing, and predicting the performance of applications involving both wavefront parallelism and black-box linear algebra kernels, and their deployment onto heterogeneous platforms. As running example, we developed and evaluated an application for signal processing in terms of General Fourier Transform (GFT).

In particular, we demonstrated how the usage of high-level tools, for both the linear algebra and the parallel coordination aspects, turns into a very limited overall coding effort (i.e., few hundred lines of code). Moreover, the proposed approach is based on a modular structure, that can be applied to different problems. In fact, the GFT kernel can be replaced with any processing to be executed on a matrix tile, while the pattern of dependencies between different tiles can be easily redefined and it is independent from the specific computation. Moreover, the proposed approach allowed to highlight and isolate a large number of factors that impacts the performance. Although tackling the same problem at a lower level (e.g., writing directly CUDA code) could lead benefits in terms of performance and control, the development effort would typically outgrow the time saved due to a more straightforward performance analysis.

On the performance side, we avoided to focus on absolute measurements, since the problem-specific optimization is out of the scope of this work. Nevertheless, the results achieved by applying the proposed approach—in particular exploiting the performance analysis it enables—are in line with ad-hoc implementations found in literature [19].

In terms of future work, we plan to apply the proposed approach to similar problems from the domain of graph-based signal processing. For instance, recent efforts in the image processing field are based on processing variable-size regions rather than fixed-size tiles. In this scenario, we envision heterogeneous deployments to be of paramount value when mapping each region size to a suitable available back-end.

ACKNOWLEDGEMENT

This research has been supported by the *Competency Center on Scientific Computing* (C3S) at University of Turin [20], by the HPC4AI project funded by the Region Piedmont POR-

FESR 2014-20 programme (INFRA-P) [21], and the OptiBike experiment in the H2020 projects Fortissimo2 (no. 680481).

References

[1] D. Shuman *et al.*, "The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains," *Signal Processing Magazine, IEEE*, vol. 30, no. 3, pp. 83–98, 2013.

[2] J. Anvik *et al.*, "Generating parallel programs from the wavefront design pattern," in *Proc. of 16th Intl. Parallel and Distributed Processing Symposium*, 2002.

[3] A. Sandryhaila and J. M. F. Moura, "Discrete signal processing on graphs: Frequency analysis," *IEEE Transactions on Signal Processing*, vol. 62, no. 12, pp. 3042–3054, 2014.

[4] F. R. Chung, *Spectral graph theory*. AMS, 1997, vol. 92.

[5] S. K. Narang, A. Gadde, and A. Ortega, "Signal processing techniques for interpolation in graph structured data," in *Proc. of IEEE Intl. Conference on Acoustics, Speech and Signal Processing*, 2013.

[6] W. Hu, G. Cheung, and A. Ortega, "Intra-prediction and generalized graph fourier transform for image coding," *Signal Processing Letters*, vol. 22, no. 11, pp. 1913–1917, 2015.

[7] C. Zhang and D. Florencio, "Analyzing the optimality of predictive transform coding using graph-based models," *IEEE Signal Processing Letters*, vol. 20, no. 1, pp. 106–109, 2013.

[8] G. Fracastoro, F. Verdoja, M. Grangetto, and E. Magli, "Superpixel-driven graph transform for image compression," in *Proc. of IEEE International Conference on Image Processing*, 2015.

[9] M. Aldinucci, S. Ruggieri, and M. Torquati, "Porting decision tree algorithms to multicore using FastFlow," in *Proc. of European Conference in Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, ser. LNCS, vol. 6321. Barcelona, Spain: Springer, Sep. 2010, pp. 7–23.

[10] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," in *Programming Multi-core and Many-core Computing Systems*, ser. Parallel and Distributed Computing. Wiley, 2017, ch. 13.

[11] E. Ayguadé *et al.*, *A Proposal for Task Parallelism in OpenMP*. Springer Berlin Heidelberg, 2008, pp. 1–12.

[12] P. Viviani, M. Torquati, M. Aldinucci, and R. d'Ippolito, "Multiple back-end support for the armadillo linear algebra interface," in *Proc. of 32nd ACM Symposium on Applied Computing*, 2017.

[13] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, 2010.

[14] G. Teodoro *et al.*, "Efficient irregular wavefront propagation algorithms on hybrid CPU-GPU machines," *Parallel Computing*, vol. 39, no. 4–5, pp. 189–211, 2013.

[15] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus," in *Proc. of Intl. Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[16] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "On the development of variable size batched computation for heterogeneous parallel architectures," in *Proc. of Intl. Parallel and Distributed Processing Symposium Workshops*, 2016.

[17] ——, "Performance tuning and optimization techniques of fixed and variable size batched cholesky factorization on GPUs," *Procedia Computer Science*, vol. 80, pp. 119–130, 2016.

[18] M. Danelutto, T. De Matteis, G. Mencagli, and M. Torquati, "A divide-and-conquer parallel pattern implementation for multicores," in *Proc. of 3rd Intl. Workshop on Software Engineering for Parallel Systems*, 2016.

[19] D. De Sensi, M. Torquati, and M. Danelutto, "A reconfiguration algorithm for power-aware parallel applications," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 4, 2016.

[20] M. Aldinucci, S. Bagnasco, S. Lusso, P. Pasteris, and S. Rabellino, "The Open Computing Cluster for Advanced data Manipulation (occam)," in *The 22nd International Conference on Computing in High Energy and Nuclear Physics (CHEP)*, San Francisco, USA, Oct. 2016.

[21] M. Aldinucci, S. Rabellino, M. Pironti, F. Spiga, P. Viviani, M. Drocco, M. Guerzoni, G. Boella, M. Mellia, P. Margara, I. Drago, R. Marturano, G. Marchetto, E. Piccolo, S. Bagnasco, S. Lusso, S. Vallero, G. Attardi, A. Barchiesi, A. Colla, and F. Galeazzi, "HPC4AI, an AI-on-demand federated platform endeavour," in *ACM Computing Frontiers*, Ischia, Italy, May 2018.