Exploiting stream parallelism of MRI reconstruction using GrPPI over multiple back-ends

Javier Garcia-Blas, David del Rio Astorga, J. Daniel Garcia and Jesus Carretero

University Carlos III of Madrid, Spain
fjblas@inf.uc3m.es

Abstract

In recent years, on-line processing of data streams has been established as a major computing paradigm. This is due mainly to two reasons: first, more and more data are generated in near real-time that need to be processed; the second reason is given by the need of efficient parallel applications. However, the above-mentioned areas expose a tough challenge over traditional data-analysis techniques, which have been forced to evolve to a stream perspective. In this work we present an comparative study of a stream-aware multi-staged application, which has been implemented using GrPPI, a generic and reusable parallel pattern interface for C++ applications. We demonstrate the benefits of using this interface in terms of programability, performance, and scalability.

Keywords MRI reconstruction, Stream parallelism, GrPPI

Formal publication `https://doi.org/10.1109/CCGRID.2019.00081`

I. Introduction

Typically, both analysis and data processing have been carried out in a batch fashion (i.e. the different computing iterations were applied over a set of stored data). However, in recent years, domains that need a set of constantly refreshed information have gained greater importance. For instance, this is the case of application fields such as scientific computing research [17], environmental research by means of sensor networks, social network analytics, and many others. In these research areas, batch techniques cannot be applied since those techniques do not meet their computing needs. The lack of effective tools for approaching this kind of problems has been solved by evolving traditional paradigm to the *stream processing* paradigm. In this paradigm, a constant flow of information retrieved from a set of sources needs to be analyzed/processed and the results must be reported in near real-time. Another outstanding need of this kind of problems is the urgency for retrieving the computed results because they either are used in decision-making processes, or are meaningful only during a limited time frame. The trend of continuous data flow processing under time constraints has been named *near real-time stream processing*. Moreover, the industry and the academic community have developed High Throughput Computing (HTC) techniques to provide solutions for these scenarios. HTC is the data-intensive variant of HPC. Fast data processing is a fundamental requirement in stream processing. To achieve a high processing throughput, there is a need to replace the batch-like typical approach to novel strategies. One of these new strategies is pipeline processing, consisting in reading the incoming data and processing it across the successive user-defined stages.

The main contribution of this work is a comparative study of the generic interface for parallel patterns, namely GrPPI, under a real use case. By using performance and hardware metrics, we compare a stream-based medical imaging application under different aspects such as CPU time, cache efficiency, and memory consumption. The rest of the paper is structured as follows. Section II revisits related works that intersect with the contributions presented in this paper. Section III introduces the GrPPI, a generic interface to parallel patterns. Section IV presents the application use case analyzed in this paper. In Section V, we discuss about experiments carried out. Section VI closes the paper with a few concluding remarks.

II. Related work

In recent years, several engines for shared-memory and distributed Autonomic Solutions for Parallel and Distributed Data Stream have been developed, such as Storm [3], Spark [19], Flink [14] and StreamIt [18]. Basically, applications implemented using these engines are represented as directed flow graphs, where nodes represent operators and the edges the stream flow. According to how the operators, or nodes in the graph, and the edges are defined, different and complex operations for filtering, splitting and joining streams can be performed.

On the other hand, a number of parallel-pattern interfaces have emerged oriented to: *i)* multi-core processors, e.g., Intel Thread Building Blocks (TBB) [15], FastFlow [2], RaftLib [4] or Kanga [13]; *ii)* heterogeneous architectures, such as, SkePU [8], which allows hybrid CPU–GPU configurations; and *iii)* distributed platforms, e.g., the Münster Skeleton Library (Muesli) [9] and CnC [6]. Simultaneously, standardized interfaces are being progressively developed. This is the case of ISO C++ Standard Parallel STL published as technical specification [11] and now part of C++17 [1]. Similar implementations to the parallel STL can also be found as third-party libraries, as HPX [12].

III. Background

In this section, we introduce the pipeline and farm streaming parallel patterns as well as the GrPPI interface to those patterns.

III.1 Streaming parallel patterns

In general, DaSP applications can be seen as data-flows in the form of directed acyclic graphs (DAG), where the source node (*producer*) gets items from some input stream, intermediate nodes perform some transformation on them, and a sink node (*consumer*) dumps transformed items to an output stream. To accelerate these applications, the nodes can be executed in parallel as long as data item dependencies are preserved. A common and simple DAG construction in DaSP is the *pipeline*, where the nodes in a topological ordering have data item dependencies only with the previous node. Another common construction is the farm pattern, where the transformation in a node is replicated n times to increase its throughput. This allows for multiple stream items to be computed in parallel. The formal definitions of the pipeline and farm patterns are the following:

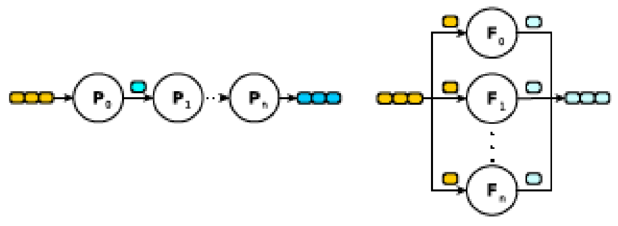
- Pipeline. Items from the input stream are processed in several parallel stages. Each stage processes data produced by the previous stage and delivers results to the next one. Provided that the i-th stage in a n-staged pipeline computes the function $f_i : \alpha \rightarrow \beta$, the pipeline delivers the item x_i to the output stream applying the function $f_n(f_{n-1}(\ldots f_1(x_i)\ldots))$ (i.e. function composition). The main requirement of this pattern is that the transformations in the stages should be independent from each other, i.e., they can be computed in parallel without side effects. The parallel implementation of this pattern is performed by using a set of concurrent entities, each of them taking care of a single stage. Figure 1(a) shows the pipeline diagram.

- Farm. Transformation $f : \alpha \rightarrow \beta$ is computed in parallel over all items from the input stream. Two items x_i and x_j can be transformed in parallel producing $f(x_i)$ and $f(x_j)$. Consequently, transformation of items need to be completely independent from each other. Figure 1(b) shows the farm diagram.

Both pipeline and farm patterns can be composed to produce more efficient applications. Basically, the compositions supported between the pipeline and farm patterns are those in which the pipeline stages can be parallelized individually using the farm pattern. Thus, if a pipeline stage corresponds with a pure function, this can be computed in parallel following a farm construction. Throughout this paper, we denote the sequential stages of a pipeline with "`p`", the farm stages with "`f`" and the communication between two stages with the symbol "`|`". For instance, a pipeline comprised of 4 stages, where the second and the third are farm stages, is represented by "`(p|f|f|p)`".

III.2 GrPPI, a generic parallel pattern interface

In order to provide a generic interface to parallel patterns we have defined GrPPI[1], a generic and reusable parallel pattern interface for C++ applications [16]. This interface takes full advantage of modern



(a) pipeline pattern. (b) farm pattern.

Figure 1: pipeline and farm pattern diagrams.

C++ features, metaprogramming concepts, and generic programming to act as switch between the OpenMP, C++ threads, Intel TBB, and FastFlow parallel programming models. Its design allows users to make use of the aforementioned execution frameworks from a unified and compact interface, hiding away the complexity behind the use of different implementation mechanisms. Furthermore, the modularity of GrPPI permits to easily integrate new patterns, while combining them to arrange more complex constructs.

GrPPI accommodates a layer between the application programmer and the different programming models. In this way, GrPPI can be used to implement a wide range of existing stream-processing and data-intensive applications with relatively small effort, having as a result portable codes that can be executed on multiple platforms.

Listing 1 shows the pipeline and farm GrPPI C++ pattern generic functions. We make use of generic programming with *variadic templates* and *forwarding references* to take multiple callable entities (e.g., a functor or lambda expression) as transformations. Note as well that the first parameter specifies the execution policy that shall be used to execute the operators. This allows to separate the programming model used as a backend from the specific configuration of the computations.

Listing 1: Pipeline and farm interface

```
template <typename E, typename G, typename ... Ts>
void pipeline(const E & execution, G && generator, Ts && ... transformers);

template <typename T>
void farm(int replicas, T && transformer);
```

Listing 2 depicts a pipeline where the first stage invokes `read_item` to get the next item in the stream. The second stage applies to each item the transformation `f()` with four parallel replicas. The third stage applies to each item the transformation `g()`. Finally, the fourth stage invokes `write_item` which writes each item to an output stream.

Listing 2: Pipeline and farm interface.

```
parallel_execution_omp ex;
pipeline(ex,
  read_item,
  farm(4,[](auto && x) { return f(x); }),
  [](auto && x) { return g(x); },
  write_item
);
```

[1] `https://github.com/arcosuc3m/grppi`

IV. pHARDI: diffusion magnetic resonance imaging toolkit

Diffusion magnetic resonance imaging is a non-invasive technique capable of quantifying the diffusion process of water molecules in living biological tissues. Its main application is the study of the local geometry and wiring pattern of the human brain white matter. A large number of neuroscience research studies and clinical applications have been conducted in the last decade [7]. Many of these studies are based on different intra-voxel (volumetric pixel) models of molecular diffusion, which in turn, require different sampling schemes to collect the data and fitting algorithms.

In order to facilitate the widespread use of the molecular diffusion technique, in a previous work, we have developed a novel toolkit called **pHARDI**[2]. The purpose of pHARDI is twofold: (1) to provide in a single toolkit with an extensive and diverse set of reconstruction methods for different sampling protocols and (2) to accelerate the reconstruction process by means of high quality linear algebra libraries. The toolkit has a layer-based design enabling the paralelization of the computation stages via multiple accelerators in a wide range of devices, including co-processors, multi-core CPU, and GPU devices. Experimental evaluation shows that pHARDI attains, on average, a speed-up of 8× over equivalent Matlab implementations [10]. Figure 2 shows the five stages in which the pHARDI implementation is structured:

- Stage 1: an initial transformation from 4D volume represented in NIfTI to a matricial format.
- Stage 2: this stage reduces the computation by applying a mask over the white matter region of the brain. After that, resulting voxels are transformed into a matrix, which considers the directions of each track. The size of this matrix determines the computational cost for future tasks.
- Stage 3: it is the most time consuming part. This stage reconstructs each slice of the volume in all the directions provided in the input files.
- Stage 4: this stage aggregates the partial results (i.e., slices) into a final reconstructed volume. Each slice can be included in the final volume without interference (embarrassing parallelism).
- Stage 5: this final step is in charge of transforming the resulting matrix into a final NIfTI representation.

Stages 2, 3 and 4 can be represented as a *pipeline*, in which each stage of the pipeline pattern can be executed independently. Additionally, Stage 3 can be accelerated using a *farm* pattern as it is computationally more expensive than the others. In this work, we demonstrate the advantage of stream processing in this multi-staged algorithm by using GrPPI.

V. Experimental evaluation

The evaluation has been carried out on a machine consisting of two multi-core Intel Xeon E5-2630 v3 processor with a total of 8 physical cores running at 2.40 GHz, hyper-threading activated, equipped with 128 GB of RAM, and executing Linux Ubuntu 16.04 x64 OS. The compilers used is GCC 7.0. After that, the source code has been compiled using both `-O3` and `-DNDEBUG` flags. In all the cases, we show the metrics for an execution with up to 20 threads.

We run the toolkit using a real diffusion MRI dataset[3] acquired from healthy subjects. Specifically, whole-brain HARDI data were acquired in a 3T Philips Achieva scanner (Sant Pau Hospital, Barcelona) with a 8-channel head coil along 100 different gradient directions on the sphere in q-space with constant $b = 2000\,\mathrm{s/mm}^2$. Additionally, $1 b = 0$ volume was acquired with in-plane resolution of $2.0 \times 2.0\,\mathrm{mm}^2$ and slice thickness of 2 mm. The acquisition was carried out without undersampling in the k-space (i.e., $R = 1$). The final dimension of this dataset is $128 \times 128 \times 60 \times 101$ voxels, resulting in a file of 117 MB.

V.1 Performance metrics analysis

This section summarizes the performance metrics analysis carried out for the pHARDI application. In this section multiple comparative metrcis are shown. We compare the sequential baseline implementation with a manual accelerated version implemented by us with OpenMP (OpenMP in tables) and GrPPI under different back-ends, such as sequential (SEQ), OpenMP (OMP), Intel TBB (TBB), C++ native threads (THR), and FastFlow (FastFlow).

We have employed *perf* [5] for collecting both execution time and performance counters. Table 1 summarizes the metrics collected for this work and its description.

V.1.1 Execution time

We observe from the PFRTI metric that GrPPI versions are as competitive as the OpenMP-based version. Besides, there are also project-based speed metrics such as PPRTI, PPSTI and PPUTI, which were measured with *time* utility from the Linux platform. The results obtained from 20 measurements were averaged (the method was the same for every other run-time metrics as well). The final values of the metrics are shown in Table 2. The table

PFRTI and PPRTI clearly indicate a reduction in execution time for the parallel part of the code as well as for the whole application. PPSTI and PPUTI refer to the total time consumed by the process. We also highlight the value 17.86 obtained for TBB-based backend, which is motivated by the fact that Intel TBB is task based and its scheduler maps those task to available hardware threads with a work stealing policy.

V.2 Performance analysis

In this subsection, we evaluate the overall execution time and speed-up reached by a refactorized version of the ported implementation of pHARDI. It is important to mention that in case of Intel TBB and FastFlow, we have turned off the unneeded cores at system level [4]. However, it is difficult to predict the interferences between both pipeline and farms threads.

Figure 3 plots the total execution time of pHARDI, comparing different back-ends of GrPPI and OpenMP. We evaluate an increasing number of farm threads of Stage 3 of the pipeline. We can observe

[2] Available at `https://github.com/arcosuc3m/phardi`.

[3] https://zenodo.org/record/1194253.Ws86V9vgKcY

[4] *echo 0 > /sys/devices/system/cpu/cpu0/online*

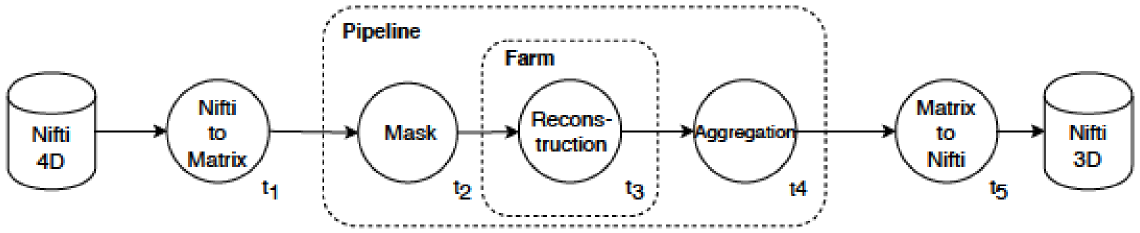


Figure 2: Parallel patterns identified in pHARDI.

Table 1: Performance and hardware counters employed is this study.

Metric	Description
PFRTI	Manually instrumented execution time.
PPRTI	Ellapsed real (wall clock) time used by the process.
PPSTI	Total number of CPU-seconds used by the system.
PPUTI	Total number of CPU-seconds that the process used directly in user mode.
PPMRS	Maximum resident set size of the process.
PPVCS	Number voluntary context switches.
PPICS	Number involuntary context-switches.
PPCRF	Number of cache references.
PPCMS	Number of cache misses.

Table 2: Metrics related to the execution time. ***M*** *represents manual paralledization while* ***A*** *represents that parralization has been carried out in an automatic way by using GrPPI.*

Implementation type	$128 \times 128 \times 60 \times 101$ **voxels [sec]**			
	PFRTI	**PPRTI**	**PPSTI**	**PPUTI**
baseline	1,120.51	1,115.20	48.87	1,596.71
OpenMP (M)	77.72	78.18	16.00	1,344.16
GrPPI + SEQ (A)	947.82	944.50	4.99	946.62
GrPPI + OMP (A)	79.43	82.12	10.18	1,399.51
GrPPI + TBB (A)	88.11	90.18	17.86	**1,233.89**
GrPPI + THR (A)	79.50	79.25	8.98	1,391.80
GrPPI + THR No Order (A)	**77.04**	**77.39**	**7.94**	1,383.30
GrPPI + FastFlow (A)	143.33	143.38	8.26	1,765.88

that GrPPI scales with an increasing number of threads, reaching its minimum execution time with 16 threads. This number corresponds with the number of physical cores of the employed machine.

Equations 1, 2, and 3 depict the theoretical sequential and parallel execution times. N corresponds with the number of slices. K represents the number of active cores involved in the parallel execution. $t1$ to $t5$ correspond with the stage in the defined pipeline.

Given the limitations of transforming NIfTI-based data, Stages 1 and 5 run sequentially in both sequential and parallel implementations. Due to the use of the pipeline parallel pattern, we have experimentally characterized the execution times of Stages 2 and 4 and we conclude that those stages can be ignored.

$$T_{SEQ} = t_1 + N \times (t_2 + t_3 + t_4) + t_5 \tag{1}$$

$$T'_{SEQ} = t_1 + t_5 + N \times max(t_2 + t_3 + t_4) \simeq t_1 + t_5 + N \times t_3 \tag{2}$$

$$T_{PAR} = t_1 + t_5 + N \times max(t_2 + \frac{t_3}{K} + t_4) \simeq t_1 + t_5 + \frac{N}{K} \times t_3 \tag{3}$$

$$S = \frac{t_1 + t_5 + N \times t_3}{t_1 + t_5 + \frac{N}{K} \times t_3} \tag{4}$$

Figure 4 plots the achieved speed-up of pHARDI under different back-ends compared with the baseline implementation based on sequential C++. The figure shows the theoretical speed-up calculated by using Equation 4. We highlight that most of the back-ends outperform the theoretical maximum speed-up. This is mainly due to the cache behavior under the parallelized scenario. GrPPI's back-ends are not affected by the increment of concurrent threads, maintaining the values of metrics like number of L1 data cache

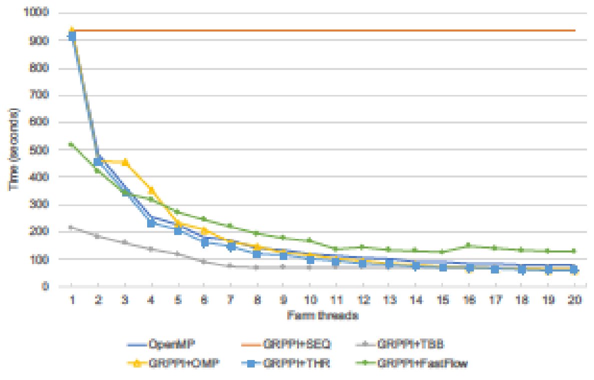


Figure 3: Performance evaluation over five GRPPI*'s back-ends and a manually implemented version of pHARDI using OpenMP.*

misses. Furthermore, the increase of hit ratio in L2 and L3 motivates the performance over the theoretical speed-up (black line).

We can observe also that the reached speed-up of the OpenMP version is close to the theoretical metric. However, GRPPI-based versions outperform this theoretical bound due to a higher hit rate in both L2 and L3 cache levels.

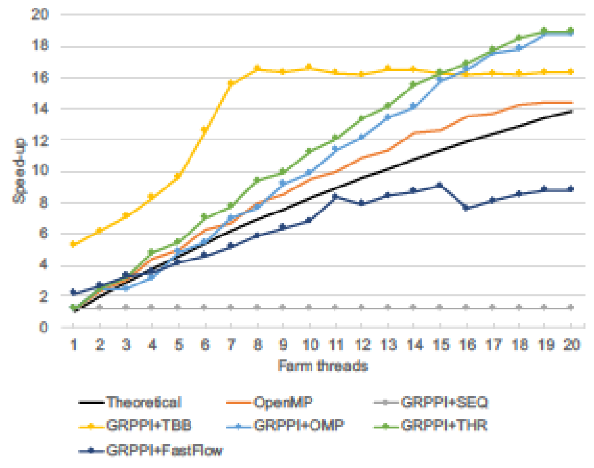


Figure 4: Performance evaluation over five GRPPI*'s back-ends of PHARDI and its speed-up compared with the baseline solution.*

VI. Conclusion

Stream processing frameworks are current trends in the data processing field. An evidence of this fact is the growing number of solutions in both industry and academia. In this work we have shown the benefits of using the GRPPI generic API in a real medical image processing application. The evaluation results demonstrate the acceleration obtained after applying refactoring techniques, clearly outperforming the baseline version. This has allowed us to accelerate the processing time by a factor of 18.

VII. Acknowledgment

This work was supported by the EU project "ASPIDE: Exascale Programing Models for Extreme Data Processing" under grant 801091.

References

[1] ISO/IEC 14882:2017 – Programming Languages – C++. International standard, ISO/IEC (Dec 2017)

[2] Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. Programming multi-core and many-core computing systems, parallel and distributed computing (2014)

[3] Allen, S.T., Jankowski, M., Pathirana, P.: Storm Applied: Strategies for Real-time Event Processing. Manning Publications Co., Greenwich, CT, USA, 1st edn. (2015)

[4] Beard, J.C., Li, P., Chamberlain, R.D.: RaftLib: A C++ Template Library for High Performance Stream Parallel Processing. In: Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores. pp. 96–105. PMAM '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2712386.2712400

[5] Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters. In: Supercomputing, ACM/IEEE 2000 Conference. pp. 42–42. IEEE (2000)

[6] Budimlić, Z., Burke, M., Cavé, V., Knobe, K., Lowney, G., Newton, R., Palsberg, J., Peixotto, D., Sarkar, V., Schlimbach, F., Taşirlar, S.: Concurrent Collections. Sci. Program. **18**(3-4), 203–217 (Aug 2010). https://doi.org/10.1155/2010/521797

[7] Canales-Rodríguez, E.J., Daducci, A., Sotiropoulos, S.N., Caruyer, E., Aja-Fernández, S., Radua, J., Mendizabal, J.M.Y., Iturria-Medina, Y., Melie-García, L., Alemán-Gómez, Y., et al.: Spherical deconvolution of multichannel diffusion MRI data with non-Gaussian noise models and spatial regularization. PloS one **10**(10), e0138910 (2015)

[8] Enmyren, J., Kessler, C.W.: SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In: Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications. pp. 5–14. HLPP '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1863482.1863487

[9] Ernsting, S., Kuchen, H.: Data Parallel Algorithmic Skeletons with Accelerator Support. International Journal of Parallel Programming **45**(2), 283–299 (Apr 2017). https://doi.org/10.1007/s10766-016-0416-7

[10] Garcia-Blas, J., Dolz, M.F., Garcia, J.D., Carretero, J., Daducci, A., AlemÃąn, Y., Canales-RodrÃ■guez, E.J.: Porting Matlab applications to high-performance C++ codes: CPU/GPU-accelerated spherical deconvolution of diffusion MRI data. In: ICA3PP: 16th International Conference on Algorithms and Architectures for Parallel Processing. pp. 630–643 (December 2016)

[11] ISO/IEC: Programming Languages – Technical Specification for C++ Extensions for Parallelism (Jul 2015), ISO/IEC TS 19570:2015

[12] Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D.: HPX: A Task Based Programming Model in a Global Address Space. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models. pp. 6:1–6:11. PGAS '14, ACM, New York, NY, USA (2014)

[13] Kist, D., Pinto, B., Bazo, R., Bois, A.R.D., Cavalheiro, G.G.H.: Kanga: A Skeleton-Based Generic Interface for Parallel Programming. In: 2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW). pp. 68–72 (Oct 2015)

[14] Papp, S.: The Definitive Guide to Apache Flink: Next Generation Data Processing. Apress, Berkely, CA, USA, 1st edn. (2016)

[15] Reinders, J.: Intel threading building blocks - outfitting C++ for multi-core processor parallelism. O'Reilly (2007)

[16] del Rio Astorga, D., Dolz, M.F., Fernández, J., García, J.D.: A Generic Parallel Pattern Interface for Stream and Data Processing. Concurrency and Computation: Practice and Experience (Online), e4175–n/a (April 2017)

[17] Stoica, R., Frank, M., Neufeld, N., Smith, A.C.: Data Handling and Transfer in the LHCb Experiment **55**(1), 272âĂŞ277

[18] Thies, W., Karczmarek, M., Amarasinghe, S.: Streamit: A language for streaming applications. In: Horspool, R.N. (ed.) Compiler Construction. pp. 179–196. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)

[19] Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache spark: A unified engine for big data processing. Commun. ACM **59**(11), 56–65 (Oct 2016)