Received: 13 January 2017 | Revised: 23 March 2017 | Accepted: 21 April 2017
DOI: 10.1002/cpe.4175

WILEY

SPECIAL ISSUE PAPER

A generic parallel pattern interface for stream and data processing

David del Rio Astorga | Manuel F. Dolz | Javier Fernández | J. Daniel García

Computer Science and Engineering Department, University Carlos III of Madrid, 28911–Leganés, Spain

Correspondence
David del Rio Astorga, Computer Science and Engineering Department, University Carlos III of Madrid, 28911–Leganés, Spain.
Email: david.rio@uc3m.es

Funding information
EU project, ICT 644235 "REPHRASE: REfactoring Parallel Heterogeneous Resource-Aware Applications" and Spanish "Ministerio de Economía y Competitividad", Grant/Award Number: TIN2016-79673-P; "Towards Unification of HPC and Big Data Paradigms"

Summary

Current parallel programming frameworks aid developers to a great extent in implementing applications that exploit parallel hardware resources. Nevertheless, developers require additional expertise to properly use and tune them to operate efficiently on specific parallel platforms. On the other hand, porting applications between different parallel programming models and platforms is not straightforward and demands considerable efforts and specific knowledge. Apart from that, the lack of high-level parallel pattern abstractions, in those frameworks, further increases the complexity in developing parallel applications. To pave the way in this direction, this paper proposes GRPPI, a generic and reusable parallel pattern interface for both stream processing and data-intensive C++ applications. GRPPI accommodates a layer between developers and existing parallel programming frameworks targeting multi-core processors, such as C++ threads, OpenMP and Intel TBB, and accelerators, as CUDA Thrust. Furthermore, thanks to its high-level C++ application programming interface and pattern composability features, GRPPI allows users to easily expose parallelism via standalone patterns or patterns compositions matching in sequential applications. We evaluate this interface using an image processing use case and demonstrate its benefits from the usability, flexibility, and performance points of view. Furthermore, we analyze the impact of using stream and data pattern compositions on CPUs, GPUs and heterogeneous configurations.

KEYWORDS
high-level API, parallel pattern, parallel programming framework, stream processing

1 | INTRODUCTION

Compared with sequential programming, designing and implementing parallel applications for operating on modern hardware poses a number of new challenges to developers.[1] Communication overheads, load imbalance, poor data locality, improper data layouts, contention in parallel I/O, deadlocks, starvation, or the appearance of data races in threaded environments are just examples of those challenges. Besides, maintaining and migrating such applications to other parallel platforms demands considerable efforts. Thus, it becomes clear that programmers require additional expertise and endeavor to implement parallel applications, apart from the knowledge needed in the application domain.

An approach to relieve developers from this burden is the use of pattern-based parallel programming frameworks, such as SkePU,[2] FastFlow[3], or Intel TBB.[4] In this sense, design patterns provide a way to encapsulate (using a building blocks approach) algorithmic aspects, allowing users to implement more robust, readable, and portable solutions with such a high-level of abstraction. Basically, these patterns instantiate parallelism while hide away the complexity of concurrency mechanisms, eg, thread management, synchronizations, or data sharing. Examples of applications coming from multiple domains (eg, financial, medical, and mathematical) and improving their performance through parallel programming design patterns, can be widely found in the literature.[5-7] Nevertheless, although all these skeletons aim to simplify the development of parallel applications, there is not a unified standard.[8] Therefore, users require understanding different frameworks, not only to decide which fits best for their purposes, but also to properly use them. Not to mention the migration efforts of applications among frameworks, which becomes as well an arduous task.

In order to mitigate this situation, this paper presents GRPPI, a generic and reusable high-level C++ parallel pattern interface that comprises both stream and data-parallel patterns. In general, the goal of GRPPI is to accommodate a layer between developers and existing parallel programming frameworks targeted to multi-core and heterogeneous platforms. Basically, GRPPI allows users to implement parallel applications without having a deep understanding of existing parallel programming frameworks or third-party interfaces and, thus, relieves the development and maintainability efforts. In contrast to other object-oriented implementations in the literature, we use C++ template meta-programming techniques in order to provide generic interfaces of the patterns without incurring in significant runtime overheads. Specifically, we contribute in this paper with the following:

- We present a generic, reusable set of parallel pattern for the C++ language that interface with different parallel programming frameworks targeted to multi-core processors, such as C++ threads, OpenMP and Intel TBB, and accelerators, such as CUDA Thrust.
- We provide support for stream processing (Pipeline, Farm, Filter and Accumulator) and data (Map, Reduce, Stencil, MapReduce and Divide&Conquer) parallel patterns.
- We show the flexibility and the composability of GRPPI for both stream and data patterns, and their combination through diverse simple examples.
- We evaluate the overheads introduced by the interface using a real-world image processing application with regard to other pattern-based parallel frameworks and runtime environments.
- We analyze the impact of using different stream and data pattern compositions on CPU, GPUs and heterogeneous configurations on the aforementioned use case.

In general, this paper extends the results presented in[9] with *i)* the inclusion of data parallel patterns in GRPPI, *ii)* the support for accelerators, and *iii)* the pattern composition analysis along with its evaluation on both multi-core and heterogeneous platforms.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of related works in the area. Section 3 states the formal definition of the stream and data parallel patterns supported by the interface. Section 4 describes the generic parallel pattern interface presented in this contribution. Section 5 evaluates overheads of the interface on several parallel programming frameworks and analyzes the pattern composition benefits on different platform configurations. Section 6 provides a few concluding remarks and outlines future works.

2 | RELATED WORK

Multiple works proposing patterns for developing applications targeted to run on modern architectures can be found in the state-of-the-art. Indeed, pattern programming has become one of the best codifying practices in software engineering.[10] The reason is clear: they simplify the application structure while achieve a good balance between maintainability and portability. In this sense, one of the most common ways to express parallelism are parallel skeletons or patterns.[11] These patterns can be classified in 2 main categories: data parallel, eg, Map, Reduce or MapReduce; and stream parallel patterns, eg, Pipeline, Farm or Filter.[12]

Most of the existing pattern-based frameworks in the literature are data-parallel computing oriented. Focusing on implementations targeted to run on multi-core processors, we find solutions such as ArBB[13] and Kanga.[14] ArBB defines a collection of basic data classes and member functions to define data-parallel skeletons, which can be used with alternative front-ends. The Kanga framework also supports task-parallel skeletons; nevertheless, it lacks of stream-processing patterns. We can also find frameworks that implement data-parallel patterns tailored for accelerators. For example, open-source approaches, like SkePU,[2] allow deploying applications to run on both multi-core CPUs and multi-GPU environments. Commercial solutions are also present in the market, eg, Thrust[15] and SYCL[16] for CUDA and OpenCL devices, respectively. In both cases, these frameworks use a C++ library similar to the Standard Template Library to ease the parallelization task. Simultaneously, standardized interfaces are being progressively developed. This is the case of the Technical Specification for C++ Extensions for parallelism,[17] which is expected to be available as part of the forthcoming C++17 standard. Similar implementations to the parallel Standard Template Library algorithms, such as HPX,[18] can also be found as third-party libraries. As observed, all these frameworks provide high-level interfaces, enabling parallelism and easing the portability among platforms. However, although they support a well-established collection of data-parallel patterns, they still lack stream processing-oriented patterns.

Focusing on libraries that support stream-processing patterns, we find a set of well-known frameworks, such as Intel Threading Building Blocks (TBB), FastFlow, and RaftLib. TBB[4] is a C++ parallel framework based on the queue-based parallelism approach. However, it runs best on Intel-based architectures and has no support for GPUs. FastFlow[3] is a skeleton programming framework that uses lock-free communication mechanisms to implement internally its parallel patterns. This approach has support for CUDA and OpenCL. Finally, RaftLib[19] is a C++ template library that aims to fully exploit the stream processing paradigm, supporting dynamic queue optimization, automatic parallelization, and real-time performance monitoring. However, all these parallel frameworks are not yet usable nor generic enough to be easily adopted by users when developing parallel applications. The contributions presented in this paper leverages novel C++ templates and metaprogramming techniques for implementing both stream parallel and data parallel paradigms in a single and high-level interface that fills the gap from current approaches.

3 | PARALLEL PATTERNS

Patterns can be loosely defined as commonly recurring strategies for dealing with particular problems. This methodology has been widely used in multiple areas, such as architecture, object-oriented programming, and software architecture.[20] In our case, we leverage patterns for parallel software design, as it has been recognized to be one of the best codifying practices.[10] This is mainly because patterns provide a mechanism to encapsulate algorithmic features, making them more robust, portable and reusable, while if tuned, they can achieve better parallel scalability and data locality. In general, parallel patterns can be categorized in 2 groups: stream parallel patterns, eg, Pipeline, Farm and Filter; and data parallel, eg, Map, Reduce and MapReduce.[12] Following this classification, we describe formally the patterns supported by our interface in the next 2 sections.

3.1 | Stream patterns

In this section, we describe formally the stream parallel patterns Pipeline, Farm, Filter, and Accumulator included in GRPPI.

Pipeline This pattern processes the items appearing on the input stream in several parallel stages (see Figure 1A). Each stage of this pattern processes data produced by the previous stage in the pipe 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 $f_n(f_{n-1}(\dots f_1(x_i) \dots))$. The main requirement of this pattern is that the functions related to the stages should be pure, ie, they can be computed in parallel without side effects.

Farm This pattern computes in parallel the function $f : \alpha \rightarrow \beta$ over all the items appearing in the input stream (see Figure 1B). Thus, for each item x_i on the input stream the Farm pattern delivers an item to the output stream as $f(x_i)$. In this pattern, the computations performed by *f* for the items in the input stream should be completely independent to each other, otherwise they cannot be processed in parallel.

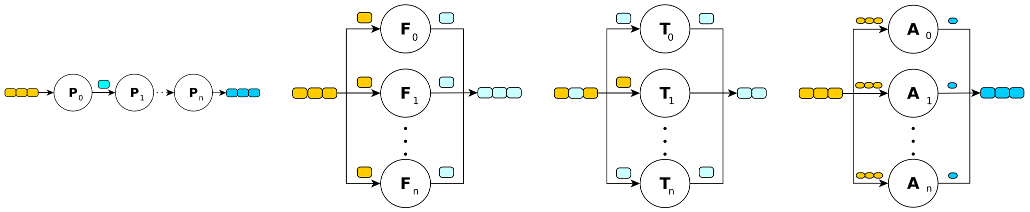
Filter This pattern computes in parallel a filter over the items appearing on the input stream, passing only to the output stream those items satisfying the boolean "filter" function (or predicate) $\mathcal{P} : \alpha \rightarrow \{true, false\}$ (see Figure 1C). Basically, the pattern receives a sequence of input items $\dots , x_{i+1}, x_i, x_{i-1}, \dots$ and produces a sequence of output items of the same type but with different cardinality. The evaluation of the filtering function on an input item should be independent to any other, ie, the predicate should be a pure function.

Accumulator This pattern collapses items appearing on the input stream and delivers these results to the output stream (see Figure 1D). The function used to collapse item values $\oplus$ should be a pure binary function of type $\oplus : \alpha \times \alpha \rightarrow \alpha$, being usually associative and commutative. Basically, the pattern computes the function $\oplus$ over a finite sequence of input items $\dots , x_{i+1}, x_i, x_{i-1}, \dots$ to produce a collapsed item on the output stream. The number of elements to be accumulated depends on the window size set as parameter.

3.2 | Data patterns

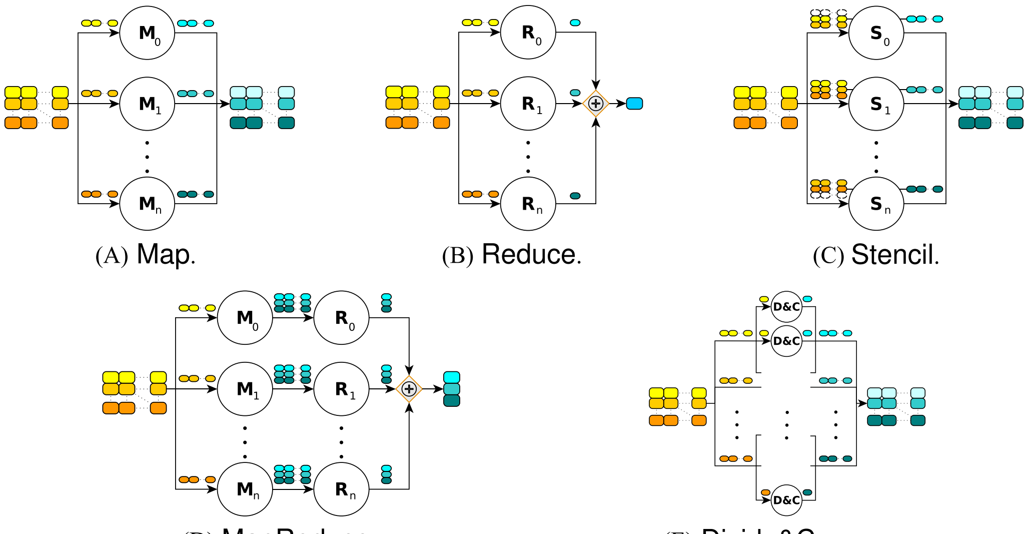
In this section, we describe formally the data parallel patterns Map, Reduce, Stencil, MapReduce and Divide&Conquer provided by GRPPI.

Map This data parallel pattern computes the function $f : \alpha \rightarrow \beta$ over the elements of the input data collection, where the input and output elements are α and β types, respectively (see Figure 2A). The output result is the collection of elements $y_1, y_2, \dots , y_N$, where $y_i = f(x_i)$ for each $i = 1, 2, \dots , N$ and x_i is the *i*-th element of the input collection. The only requirement of the Map pattern is that the function *f* should be pure.



(A) Pipeline. (B) Farm. (C) Filter. (D) Accumulator.

FIGURE 1 Stream parallel patterns



(A) Map. (B) Reduce. (C) Stencil. (D) MapReduce. (E) Divide&Conquer.

FIGURE 2 Stream parallel patterns

Reduce This data parallel pattern aggregates the elements of the input data collection of type α using the binary function $\oplus : \alpha \times \alpha \rightarrow \alpha$, that is usually associative and commutative. Finally, the result of the pattern is summarized in a single element y of type α that is obtained performing the operation $y = x_1 \oplus x_2 \oplus \ldots x_N$, where x_i is the i-th data item of the input data collection (see Figure 2B). The main constraint of this pattern is that the binary function should be pure.

Stencil This pattern is a generalization of the Map pattern in which an elemental function can access, not only to a single element in an input collection, but also to a set of neighbors (see Figure 2C). The function $f : \alpha * \rightarrow \alpha$ used by the Stencil pattern receives the input item and a set of neighbors ($\alpha *$) and produces an output element of the same type. The main requirement of this pattern is that the function f should be pure.

MapReduce This pattern computes, in a first stage a Map-like pattern, a key-value function over all the elements of an input collection, and delivers, in a second stage a Reduce-like pattern, a set of unique key value pairs where the value associated to the key is the "sum" of the values output for the same key (see Figure 2D). To do so, the MapReduce pattern computes in the Map function $f : \alpha \rightarrow \{\alpha, Key\}$ the elements in the input collection; afterwards, it uses the Reduce binary function $\oplus : \beta \times \beta \rightarrow \beta$ to sum up the partial results with the same key. The result of this pattern is a collection of data elements of type β, one per key. The requirements of the MapReduce pattern is that both Map and Reduce-related functions should be pure.

Divide&Conquer This pattern computes a problem by means of breaking it down into 2 or more subproblems of the same kind until the base case is reached and solved directly. Afterwards, the solutions of the subproblems are merged to provide a solution to the original problem (see Figure 2E). In other words, this pattern applies the function $f : \alpha * \rightarrow \beta *$ on a collection of elements of type α and produces a collection of elements of type β. A divide function $\mathcal{D}$ is used first to split the collection into distinct partitions up to the size of the base problem, which can be solved directly applying f. Finally, the partial results of the base problems are combined according to a merge function $\mathcal{M}$ in order to build the final output collection. The requirements of the Divide&Conquer pattern is that the functions f, $\mathcal{S}$ and $\mathcal{M}$ should be pure.

4 | A GENERIC AND REUSABLE PARALLEL PATTERN INTERFACE

In this section, we introduce our generic and reusable parallel pattern interface (GRPPI) for C++ applications. GRPPI takes full advantage of modern C++ features, metaprogramming concepts, and generic programming to act as switch between the parallel programming models OpenMP, C++ threads, Intel TBB and CUDA Thrust. Its design allows users to leverage the aforementioned execution frameworks just in a single and compact interface, hiding away the complexity behind the use of concurrency mechanisms. Furthermore, the modularity of GRPPI permits to easily integrate new patterns, while combining them to arrange more complex ones. Thanks to this property, GRPPI can be used to implement a wide range of existing stream-processing and data-intensive applications with relative small efforts, having as a result portable codes that can be executed on multiple frameworks.

Next, we describe in detail the interfaces of the parallel patterns offered by GRPPI and demonstrate its composability through different simple examples.

4.1 | Description of the interfaces

GRPPI offers both stream patterns and data patterns with a single interface carefully designed to allow composability and to support multiple implementation back-ends.

4.1.1 | Stream patterns

The GRPPI stream parallel patterns includes the Pipeline, Farm, Filter, and Accumulator patterns.

Pipeline The GRPPI interface designed for the Pipeline pattern receives the execution model and the functions (`in` and `stages`) related to its stages. As can be seen in Listing 1, its C++ interface uses templates, making it more flexible and reusable for any data type. Note as well the use of variadic templates, allowing a Pipeline to have an arbitrary number of stages by receiving a collection of callable objects passed as arguments. In GRPPI, the parallel implementation of this pattern is performed using a set of concurrent entities, each of them taking care of a single stage. This is controlled via the execution model parameter, that can be set to operate in sequential or in parallel, through the different supported frameworks; eg, to use OpenMP, the parameter should be set to `parallel_execution_omp`.

Farm In a similar way, the Farm pattern interface, shown in Listing 2, receives the execution model and 3 functions (`in`, `farm` and `out`) that are in charge of *(i)* consuming the items from the input stream, *(ii)* processing them individually, and *(iii)* delivering the results to the output stream. Note that the `farm` function will be executed in parallel by the different concurrent entities. In this case, the execution model can optionally receive, as an argument, the number of entities to be used for the parallel execution, eg, `parallel_execution_omp{6}` uses 6 OpenMP

Listing 1: Pipeline interface.

```
template <typename ExecMod, typename InFunc, typename ... Arguments>
void Pipeline( ExecMod m, InFunc const &in, Arguments ... stages );
```

Listing 2: Farm interface.

```
template <typename ExecMod, typename InFunc, typename TaskFunc, typename OutFunc>
void Farm( ExecMod m, InFunc const &in, TaskFunc const &farm, OutFunc const &out );
```

worker threads. If this argument is not given, the interface takes by default the number of threads set by the underlying platform.

Filter The interface for the Filter pattern, described in Listing 3, receives the execution model argument, followed by a stream consumer (`in`), filter (`filter`) and producer (`out`) functions. Specifically, the `in` function reads items from the input stream and forwards them to the `filter` function, which is responsible to determine whether an item should be accepted or not. Afterwards, those items that satisfy the filtering routine, are received by the `out` function in order to deliver them to the output stream. Note that it is mandatory that the `filter` function to return a boolean expression. The parallel implementation of this pattern applies the filter function using a set of concurrent entities, that can be configured in the execution model parameter.

Accumulator The Accumulator pattern aims at reducing, using a specific reduction (`redop`) function, the items appearing on the input stream. Similar to the other interfaces, the Accumulator interface, as shown in Listing 4, receives the execution model; the stream consumer (`in`) function; the window size, ie, the number of items that will be part of each reduction operation; the offset, determining the number of overlapping items among windows; the reduction operator; and a producer (`out`) function, responsible for delivering the items to the output stream. In this case, the concurrent entities in the parallel implementation are responsible for processing individually the accumulation of the input stream windows.

4.1.2 | Data patterns

This section describes in detail the interfaces for the data parallel patterns Map, Reduce, Stencil, MapReduce and Divide&Conquer supported by GRPPI.

Map The GRPPI interface for the Map pattern, shown in Listing 5, receives the following input parameters: the execution model, references to the first and last elements of the input data collections and the kernel (`map`) function. After the computation, the result of the Map pattern is left in the corresponding position of the output data set. Given that each element in the input data collection is independent to each other, the parallel execution of the Map pattern can be performed in the following way. First, the input collection is divided equally among the available concurrent entities. Afterwards, these entities execute in parallel the kernel `map` function and write the results in the corresponding segments of the output data collection.

Reduce The interface for the Reduce pattern, as described in Listing 6, takes the execution model, a reference to the first and last elements of the input data collection and the reduce operator. The result of the reduction is written in the output parameter passed by reference. According to the properties of the reduce operator, the reduce computation can be performed in parallel. Thus, the input data collection is partitioned in *N* chunks and computed in parallel by *N* different concurrent entities that produce a set of partial results. Finally, the result of the Reduce pattern is calculated in series by one of these entities.

Stencil The GRPPI interface for the Stencil pattern, presented in Listing 7, is quite similar to that for the Map pattern, with the exception that it additionally receives the neighborhood (`nh`) function. This function is responsible for accessing the neighbors in a given coordinate of the input data set. The parallel implementation of the Stencil pattern is analogous to that for the Map pattern. However, accessing the neighbors in the boundaries of a partitioned data set might require additional comparisons between the positions of the elements.

Listing 3: Filter interface.

```
template <typename ExecMod, typename InFunc, typename FilterFunc, typename OutFunc>
void Filter( ExecMod m, InFunc const &in, FilterFunc const &filter, OutFunc const &out );
```

Listing 4: Accumulator interface.

```
template <typename GenFunc, typename ReduceOperator, typename OutFunc>
void Accumulator( ExecMod m, GenFunc const &in, int windowsize, int offset, ReduceOperator const
     &redop, OutFunc const &out );
```

Listing 5: Map interface.

```
template <typename ExecMod, typename InputIt, typename OutputIt, typename TaskFunc, typename ...
     MoreIn>
void Map(ExecMod m, InputIt first, InputIt last, OutputIt firstOut, TaskFunc const &map, MoreIn
     ... inputs );
```

Listing 6: Reduce interface.

```
template <typename ExecMod, typename InputIt, typename Output, typename ReduceOperator>
void Reduce(ExecMod m, InputIt first, InputIt last, Output &out, ReduceOperator const &redop);
```

Listing 7: Stencil interface.

```
template <typename ExecMod, typename InputIt, typename OutputIt, typename TaskFunc, typename
     NFunc, typename ... MoreIn>
void Stencil(ExecMod m, InputIt first, InputIt first, InputIt last, OutputIt firstOut, TaskFunc
     const &stencil, NFunc const &nh, MoreIn ... inputs);
```

MapReduce The interface for the MapReduce pattern combines internally calls to the Map and ReduceGRPPI pattern interfaces. As for input parameters, it receives the execution model, references to the first and last elements of the input data collections, the kernel (`map`) function and the reduction operator for the Reduce pattern. The result is finally left in a reference to the first element of the output collection. The parallel implementation of this pattern in GRPPI exploits the parallelism offered internally by the Map and Reduce parallel patterns. The result of the Map operation is then shuffled and reduced in parallel, one for all elements with the same key. The global result for each key is finally reduced in series by one of the concurrent entities.

Divide&Conquer The interface designed for the Divide&Conquer pattern consists of the following elements: the execution model, a reference of the input data collection and the `divide`, `base_case` and `merge` functions. The result of this pattern is written to the output data collection passed by reference. The parallel implementation of this pattern in the GRPPI interface leverages first the divide kernel to steadily split the problem into smaller ones. This operation is performed by the available concurrent entities until the minimal problem dimension is reached and where the base-case solution kernel is applied. Taking the partial solutions generated, the concurrent entities merge the results in a tree-based structure until the global solution is obtained. Note that because the tree width can grow above the maximum number of concurrent entities specified, a pool of tasks is used instead in order to implement a dynamic scheduling approach.

4.2 | Pattern composability

As mentioned in the introduction, the patterns offered by GRPPI can be composed among them to produce more complex structures and to match specific constructions present in both stream and data parallel applications. To demonstrate this feature, we describe 3 examples of pattern composability tackling each of the feasible combinations of computational paradigms (stream and data) supported by GRPPI interface: stream-stream, data-data, and stream-data compositions.

For the stream-stream pattern composability, the code in Listing 10 implements a Pipeline in which the second stage is a Farm pattern. The Pipeline stages, passed as lambda functions, perform the following tasks: (i) read the lines of an input file with blank-separated values and pack them into a vector structure; (ii) compute the maximum value from incoming vectors using the Farm pattern; and (iii) print the maximum values of the vectors onto an output stream. Given that the Pipeline receives the OpenMP parallel execution model (line 1), the stages are computed in parallel by the 3 worker threads. Similarly, the nested Farm pattern is executed by 6 OpenMP threads. Note as well that `std::optional` variables, from the c++ Library Fundamentals Extensions (ISO/IEC 19568:2015), are used to mark the end of the streams with an empty value. We denote this Pipeline-Farm composition as `(p|f|p)`, being `p` and `f`, respectively, sequential and Farm-based stages. As can be seen, thanks to the use of metaprogramming techniques, templates, and lambda expressions, it is possible to easily compose GRPPI parallel patterns in order to build more complex ones.

Regarding the data-data pattern composability, Listing 11 shows a construction where a Map pattern is composed with a Reduce operation. In this case, the input matrix in the Map pattern is divided into equal partitions among the worker threads. Next, for each row in a partition, the nested Reduce pattern sums up its values and stores the result in the corresponding position of the output vector, passed as an argument in the Map function call. Note that the parallel execution model for the Map pattern is OpenMP while the nested Reduce pattern uses C++ threads, each of them using 6 worker threads. We denote this composition as `m(r)`, being `m` and `r` the Map and Reduce patterns, respectively.

Listing 8: MapReduce interface.

```
template <typename ExecMod, typename InputIt, typename Output, typename MapFunc, typename
     ReduceOperator, typename ... MoreIn>
void MapReduce(ExecMod m, InputIt first, InputIt last, Output &out, MapFunc const &map,
     ReduceOperator const &redop, MoreIn ... inputs);
```

Listing 9: Divide&Conquer interface.

```
template <typename ExecMod, typename Input, typename Output, typename DivFunc, typename TaskFunc,
      typename MergeFunc>
void DivideAndConquer(ExecMod m, Input &problem, Output &out, DivFunc const &divide, TaskFunc
     const &base_case, MergeFunc const &merge);
```

Listing 10: Example of the Pipeline-Farm composition.

```
 1 Pipeline( parallel_execution_omp,
 2    // Stage 0: read values from a file
 3    [&]() -> optional<vector<int>> {
 4       auto r = read_list(is);
 5       return ( r.size() == 0 ) ? {} : r;
 6    },
 7    // Stage 1: takes the maximum value of the vector
 8    Farm(parallel_execution_omp{6},
 9       []( vector<int> v ) {
10          return ( v.size() > 0 ) ?
11             max_element(v.begin(), v.end()) :
12             numeric_limits<int>::min();
13    }),
14    //Stage 2: prints out the result
15    [&os]( int x ) {
16       os << x << endl;
17    }
18 );
```

Listing 11: Example of the Map-Reduce composition.

```
Map( parallel_execution_omp{6},
   // Input matrix
   mat_in.begin(), mat_in.end(),
   // Vector of accumulated values from matrix rows
   vec_out.begin(),
   // Map kernel: divide matrix into rows
   [&]( auto row_in, auto sum ) {
      // Reduce kernel: Sum up the values in a matrix row
      Reduce( parallel_execution_thr{6},
         row_in.begin(), row_in.end(),
         &sum,
         std::plus<double> );
      );
   }
);
```

Listing 12: Example of the Farm-Divide&Conquer composition.

```
Farm(parallel_execution_omp(6),
   [&]() -> optional<int> { // Read values from an input file
      auto value = read_value(is);
      return ( value > 0 ) ? value : {};
   },
   [&]( int value ) { // Compute the fibonacci number using a D&C pattern
      int fibonacci = 0;
      DivideAndConquer(parallel_execution_thr(6), value, &fibonacci,
         [&](auto &value){
            std::vector< int > subproblem;
            if( v < 2 ) subproblem.push_back(value);
            else subproblem.insert(subproblem.end(), { value-1, value-2 });
            return subproblem;
         },
         [&](auto &problem, auto &partial){
            partial = ( problem == 0 ) ? 0 : 1;
         },
         [&](auto & partial, auto & out){
            out += partial;
         }
      );
      return fibonacci;
   },
   [&]( int fibonacci ) { // Print the fibonacci values
      cout << fibonacci << endl;
   }
);
```

As mentioned, we can also compose stream with data patterns. This is a feasible composition, given that the items coming from a stream can be processed themselves using a data parallel pattern. The opposite is however not feasible because the results generated in a data pattern cannot be transformed into streams and, therefore, processed using a stream processing approach. To illustrate a stream-data pattern composition, Listing 12 shows an example where a Farm stream parallel pattern is composed with a Divide&Conquer data one. In this particular case, the Farm pattern steadily reads values stored in a file and computes, for each of them, their corresponding *i*-th Fibonacci number using the Divide&Conquer pattern. Finally, the Fibonacci numbers are printed to the end user. As shown, the parallelization of the Farm is performed using 6 OpenMP threads, while the nested Divide&Conquer pattern uses 6 C++ threads. Because each of the Farm-related threads create 6 C++ nested ones, the total number of threads computing this composition is 36. This composition is denoted as `f(d)`, being `f` and `d`, Farm and Divide&Conquer patterns, respectively.

In general, Table 1 summarizes pattern compositions grouped by the 3 possible combinations of computational paradigms supported by GRPPI interface: stream-stream, data-data, and stream-data compositions. Note that rows and columns in the tables represent the outer and inner patterns involved in a given composition, respectively. We classify each specific pattern composition with one of the following 4 categories, from less to more restrictive:

Infeasible This category represents a composition that is not supported by GRPPI.

Feasible This category denotes a composition that can be implemented in GRPPI.

Irreducible This category is a feasible composition providing a useful parallel pattern that cannot be simplified any further. Note that pattern compositions falling in this category are natively supported by GRPPI.

Useful-Reducible This category is a feasible composition implementing a pattern composition that can be simplified further but that, in some cases, provides a clearer and a more readable code than its simpler equivalent.

As shown in Table 1a, the stream-stream pattern compositions involving a Pipeline and other pattern are classified as Irreducible (except those with an outer Accumulator pattern), given that it is not possible to obtain the same parallel construction using any simpler pattern. These types of compositions are natively supported in GRPPI, as shown in Listing 10. Any other composition is considered as Feasible because they can be simplified using the outer or inner pattern with an increased parallelism degree. However, these constructions do not provide any major advantage compared to the simpler construction. On the other hand, compositions containing an outer Accumulator pattern are Infeasible, as this pattern does not receive any user function to be executed in parallel.

Focusing on data-data compositions, as shown in Table 1b, constructions whose outer pattern is Map-like (Map and Stencil) are categorized as Useful-Reducible. This is because there exists a simpler equivalent using only the outer Map-like pattern. Regarding the Reduce pattern,

TABLE 1 Parallel patterns compositions in GRPPI

		Inner pattern				
		Pipeline	**Farm**	**Filter**	**Accumulator**	
		(a) Stream-stream compositions.				
Outer pattern	Pipeline	✓ (Feasible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	
	Farm	✓ (Irreducible)	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)	
	Filter	✓ (Irreducible)	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)	
	Accumulator	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)	
		Inner pattern				
		Map	**Reduce**	**Stencil**	**MapReduce**	**Divide&Conquer**
		(b) Data-data compositions.				
Outer pattern	Map	✓ (Useful-Reducible)	✓ (Useful-Reducible)	✓ (Useful-Reducible)	✓ (Useful-Reducible)	✓ (Useful-Reducible)
	Reduce	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)
	Stencil	✓ (Useful-Reducible)	✓ (Useful-Reducible)	✓ (Useful-Reducible)	✓ (Useful-Reducible)	✓ (Useful-Reducible)
	MapReduce	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)
	Divide&Conquer	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)
		(c) Stream-data compositions.				
Outer pattern	Pipeline	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)
	Farm	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)
	Filter	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)
	Accumulator	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)

it cannot be combined with any other inner one. The reasons are the same as those for the Accumulator pattern in stream-stream compositions. Other compositions whose outer pattern is MapReduce or Divide&Conquer are classified as Feasible, as they can be implemented in GRPPI although do not bring any major advantage.

Finally, stream-data compositions are summarized in Table 1c. Compositions whose outer pattern is Pipeline or Farm are denoted as Irreducible. The combination of 2 distinct parallel paradigms (stream-data) makes these compositions unique and precludes them to be simplified any further. As for compositions with an outer Filter pattern, the output cardinality of its inner pattern dictates whether the composition is Feasible or Useful-Reducible. This is because the output of the Filter function is a boolean. For instance, in a Filter-Map composition, the output cardinality of the Map pattern is equal to the input cardinality. So that, although the predicate of the Filter pattern can be implemented by transforming the output data set into a boolean, this case does not reflect a common practice. Therefore, we classify these compositions only as Feasible. On the other hand, the output cardinality of the Reduce pattern in a Filter-Reduce composition is a sole element. Thus, the Filter predicate can be easily implemented by transforming such element into a boolean. For this reason, we categorize this construct as a special case of Feasible composition. The Filter-Divide&Conquer combination is also a special case of Feasible composition, because the output cardinality of the Divide&Conquer pattern depends on the algorithm. Finally, the Accumulator pattern is not composable and, hence, classified as Infeasible.

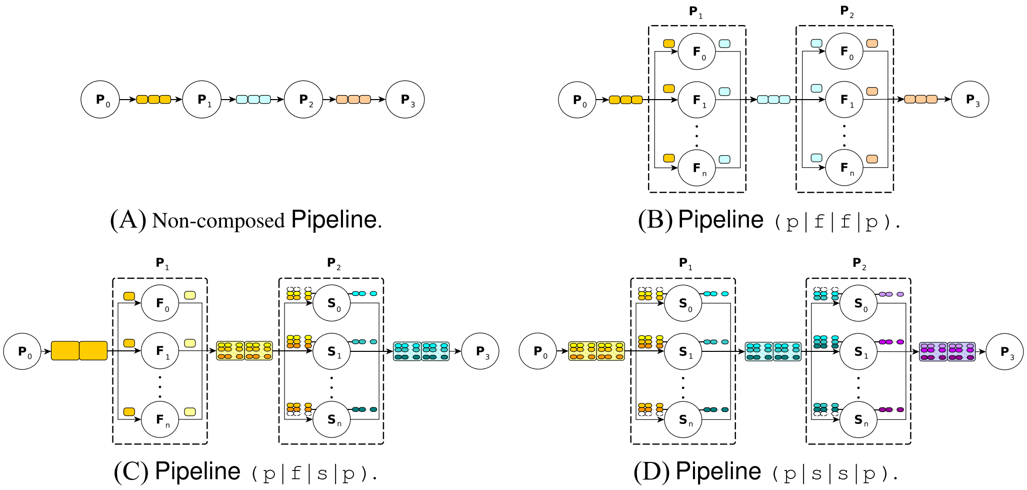
5 | EVALUATION

In this section, we perform an experimental evaluation of GRPPI in order to analyze its usability, in terms of lines of code, and its performance, in comparison with the different parallel execution environments currently supported. To do so, we use the following hardware and software components:

- *Target platform*. The evaluation has been carried out on a server platform equipped with 2× Intel Xeon Ivy Bridge E5-2630 v3 with a total of 16 cores running at 2.40 GHz, 20 MB of L3 cache and 256 GB of DDR3 RAM. This platform also incorporates a NVIDIA Tesla K40c with 12 GB and GeForce GTX680 GPUs with 2 GB of DDR5 RAM. These GPUs are denoted as `GPU0` and `GPU1`, respectively. The OS is a Linux Ubuntu 14.04.5 LTS with kernel 3.13.0-85.
- *Software*. To develop the parallel versions and to implement the proposed interfaces, we leveraged the execution environments C++11 threads and OpenMP, Intel TBB and CUDA Thrust for the GPUs. The C++ compiler used to assemble GRPPI is GNU GCC v5.0 and CUDA compiler is NVIDIA NVCC 8.0.
- *Benchmark*. To evaluate the parallel patterns, we used a video stream-processing application composed by 2 filters, the Gaussian Blur and Sobel operators. These filters are applied to an input video in order to detect edges appearing in the frames.[*] Specifically, this application matches the parallel Pipeline pattern, in which the first stage reads the frames from a video file passed as input; the second and third stages apply the Gaussian Blur and Sobel filters, respectively; and the last stage dumps the processed frames to an output video file. Note that both filters use a kernel size of 3×3. Note as well that, while the Gaussian Blur filter only performs arithmetic operations, the Sobel operator also performs a square root operation for each frame pixel processed.

To carry out the experimental evaluation, we first parallelize this video application using the above-mentioned execution frameworks and the proposed interface. Afterwards, we compare both performance and the number of lines of code required to implement such parallel versions with respect to the sequential one. To further experiment with our interface, we implemented different versions of the video application using the execution frameworks for CPUs and distinct com-

[*] This benchmark has been inspired by an OpenCV edge detection example from http://docs.opencv.org/3.1.0/d3/d63/edge_8cpp-example.html.



(A) Non-composed Pipeline.

(B) Pipeline (p|f|f|p).

(C) Pipeline (p|f|s|p).

(D) Pipeline (p|s|s|p).

FIGURE 3 Pipeline compositions of the video application

positions of patterns in its main pipeline. Specifically, we use the following Pipeline compositions: *i)* a non-composed Pipeline (p|p|p|p); *ii)* a Pipeline composed of a Farm in its second stage (p|f|p|p); *iii)* a Pipeline composed of a Farm in its third stage (p|p|f|p); and *iv)* a Pipeline composed of 2 Farm patterns in the second and third stages (p|f|f|p).

Next, we also evaluate and compare the performance when using different stream-stream and stream-data Pipeline compositions and heterogeneous configurations (CPU+GPU). In this case, we leverage different Pipeline constructions composed of: *(i)* 2 Farm patterns (p|f|f|p); *(ii)* a Farm and a Stencil in its second and third stages (p|f|s|p); *(iii)* a Stencil and a Farm in its second and third stages (p|s|f|p); and *(iv)* 2 Stencil patterns (p|s|s|p). Note that when using a Pipeline-Farm composition, each worker thread from the Farm pattern processes individual video frames. However, when leveraging a Pipeline-Stencil construction, each thread in the Stencil pattern is in charge of computing a distinct partition of a single video frame. Figure 3 illustrates some of the compositions used in these studies.

5.1 | Analysis of the usability

In this section, we analyze the usability and flexibility of the developed interface. To analyze these aspects, we compare the number of lines required to implement the parallel version of the application leveraging the interface, with respect to using directly the parallel execution frameworks. Table 2 summarizes the percentage of additional lines introduced into the sequential source code in order to implement such parallel versions for the above-mentioned pattern compositions. As can be seen, implementing more complex compositions via C++ threads or OpenMP leads to larger source codes, while for Intel TBB the number of required additional lines remains constant. Focusing on GRPPI, we observe that the effort of parallelizing an application is almost negligible: even the most complex composition increases nearly 4.4 % the number of lines of code. This behavior is contrary to the C++ threads or OpenMP frameworks, which require roughly twice of lines of code. Additionally, switching GRPPI to use a particular execution framework just needs changing a single argument in the pattern function calls.

TABLE 2 Percentage of increase of lines of code w.r.t. the sequential version

Pipeline composition	% of increase of lines of code			
	C++ Threads	OpenMP	Intel TBB	GrPPI
(p|p|p|p)	+8.8 %	+13.0 %	+25.9 %	+1.8 %
(p|f|p|p)	+59.4 %	+62.6 %	+25.9 %	+3.1 %
(p|p|f|p)	+60.0 %	+63.9 %	+25.9 %	+3.1 %
(p|f|f|p)	+106.9 %	+109.4 %	+25.9 %	+4.4 %

5.2 | Performance analysis of pattern compositions

Next, we analyze the performance with and without GRPPI using the different execution frameworks and Pipeline compositions for the video application. Concretely, we employ the frames per second (FPS) metric to analyze the behavior of the particular versions using a same input video with diverse resolutions. Also, we set the Farm stage(s) in all Pipeline compositions to be executed in parallel by 6 threads for all the execution models. Figure 4 depicts the FPS obtained for the different compositions in this experiment. A first observation is that the Pipeline combined with 2 Farm patterns for the filtering stages, in comparison with the non-composed Pipeline and compositions with only one Farm, improves substantially the FPS for all parallel frameworks. It is also remarkable that compositions using only one Farm do not bring significant improvements, because they lead to imbalanced Pipeline stages. Note that the stage running sequentially dictates the Pipeline performance, as it is the slowest one. An additional inspection into the plots reveals that the best case of Pipeline composition, which uses 2 Farm patterns, both C++11 and OpenMP deliver similar performance figures, while TBB obtains better FPS for all video resolutions. This is due to the ordering algorithm of the output stream is better optimized than those used by the other frameworks. Finally, we observe that the usage of GRPPI does not lead to significant overheads: it is less than 2 %, on average, for all the execution frameworks and compositions.

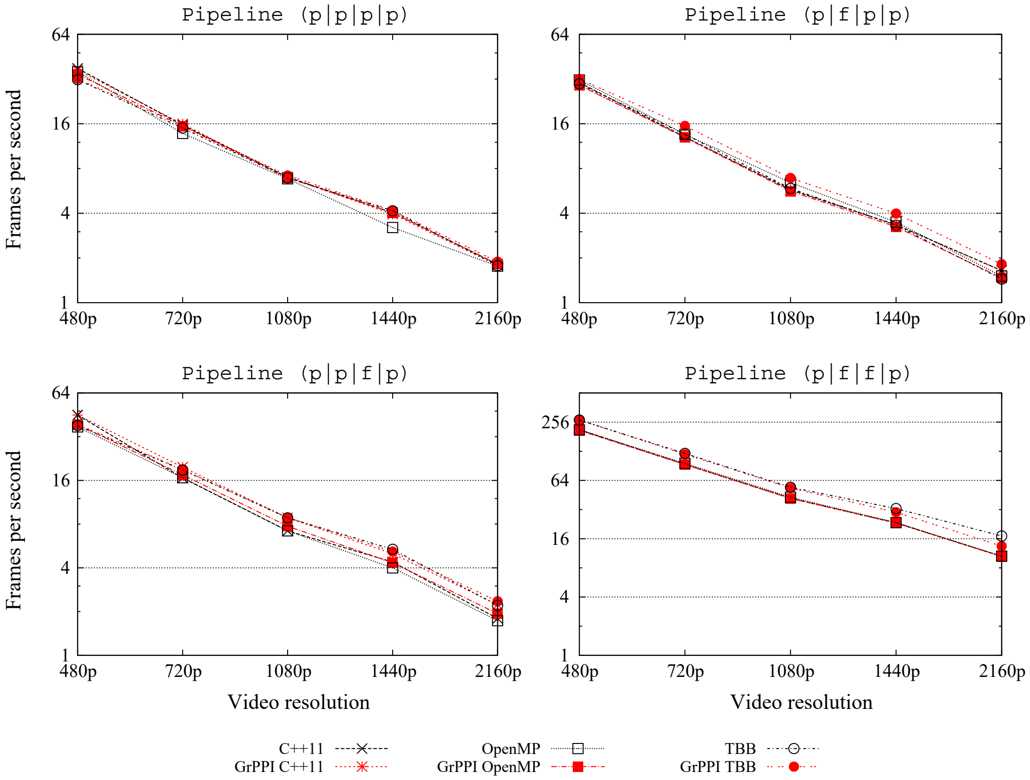


FIGURE 4 Frames per second w/ and w/o using GRPPI along with the different frameworks and Pipeline compositions

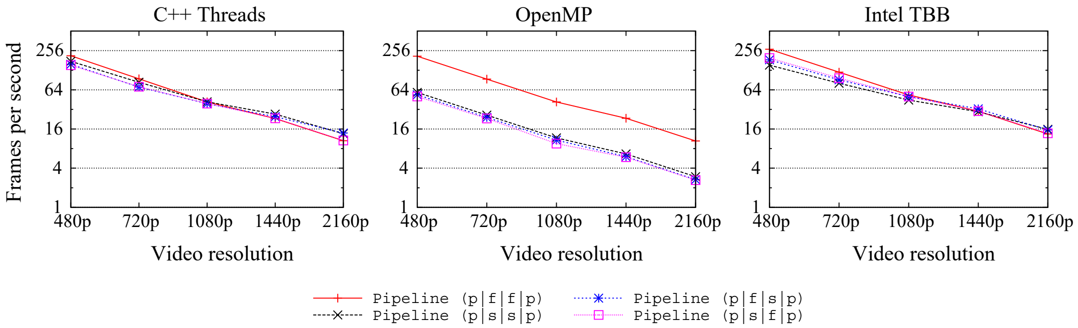


FIGURE 5 Frames per second for different frameworks and Pipeline compositions with stream and data patterns

5.3 | Performance analysis of stream vs data patterns

Our next analysis compares the performance among Pipeline compositions that combine stream and data parallel patterns. Figure 5 shows the FPS obtained for different video resolutions and parallel frameworks using GRPPI in different Pipeline compositions containing both stream and data patterns, Farm and Stencil, respectively. As can be seen, both C++ Threads and Intel TBB frameworks deliver similar performance results for all compositions. A more detailed inspection of these plots unveils an inflection point where the data-stream compositions start attaining better performance. This occurs from 1080p on for C++ Threads and from 1440p on for TBB. Note as well the slight difference using only a Stencil for computing the first or second filter. The reason behind this behavior is the higher computational load of the Sobel with respect to the Gaussian Blur filter. Regarding the OpenMP framework, it can be clearly seen that the stream-stream (p|f|f|p) composition delivers better results than using stream-data constructions. This is mainly because the worker threads in the GRPPI-Farm pattern leverage OpenMP tasks that are active during the whole video processing, while the Stencil implementation creates and destroys a task each time a video frame is processed. We figured out that the GCC-OpenMP implementation does not make use of a thread pool and, therefore, the threads in each Stencil computation are recurrently created and destroyed. Consequently, stream-data compositions in OpenMP suffer from considerable performance degradations.

5.4 | Performance analysis on heterogeneous configurations

Our last experiment, analyzes the performance of a stream-data Pipeline composition with different heterogeneous configurations (CPU+GPU). Figure 6 illustrates the FPS delivered by the Pipeline composed of 2 Stencil stages that are mapped in different ways to the devices available on the platform. As a first observation, the mapping

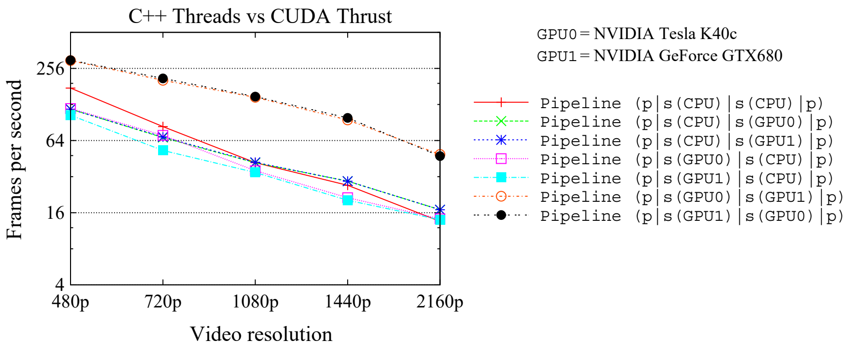


FIGURE 6 Frames per second for the Pipeline composed of 2 Stencil patterns on different heterogeneous configurations

configuration that attains the best performance is when using indistinguishably both GPUs for the Stencil stages of the Pipeline. This performance difference is due to the higher computational capacity of the GPUs, overtaking the CPUs in terms of number of cores and Single instruction, multiple data (SIMD) capabilities. Note that, in this specific use case, both arrangements of the `GPU0` and `GPU1` executing the Gaussian Blur and Sobel filters attain comparable FPS. Our next observation focuses on the configurations in which, at least, one Stencil stage is mapped to the CPU cores. In these cases, when the slowest Pipeline stage (ie, the Sobel operator) runs on a GPU, the total Pipeline throughput improves, as it contributes to have more balanced stages. Nevertheless, this advantage only applies for large frame resolutions starting from 1080p, where the data transfers overheads (host-device) compensate the amount of computation performed by the GPUs. On the other hand, if the Gaussian Blur is mapped to one of the GPUs, the throughput is limited by the Sobel operator that, executed on the CPU cores, acts as a bottleneck. Also, in this specific case, host-device data transfers do not pay off the performance improvements with respect to mapping the Gaussian Blur filter on the CPU cores.

6 | CONCLUSIONS

In this paper, we have presented a generic and reusable parallel pattern interface, namely GRPPI, which leverages modern C++ features, metaprogramming concepts, and template-based programming to act as switch between parallel programming models. Its compact design facilitates the development of parallel applications, hiding away the complexity behind the use of concurrency mechanisms. In this version of the interface, we target both stream and data parallel patterns and demonstrate its flexibility composing them on a series of simple examples. We also support frameworks targeted to multi-core processors (C++ threads, OpenMP and Intel TBB) and accelerators (CUDA Thrust). Therefore, given that many general purpose sequential applications can be decomposed in several design patterns, this interface can be easily introduced in such applications to parallelize them and improve their performance.

As observed throughout the evaluation with a parallel video application, the performance attained by each combination of parallel pattern using the supported frameworks directly with respect to using GRPPI, is almost the same. We prove as well that our approach does not lead to considerable overheads while permits to easily parallelize applications by adding, on average, 4.4% of lines of code. We also demonstrate that, depending on the application and devices available, different stream-data compositions and frameworks may lead to better performance figures. In a nutshell, GRPPI advocates for a usable, simple, generic, and high-level parallel pattern interface, allowing users to implement parallel applications without having a deep understanding of existing parallel programming frameworks or third-party interfaces.

As future work, we plan to extend GRPPI for supporting more complex parallel patterns, such as windowed and keyed stream farms, stream iteration and the Stencil-Reduce[21] patterns. Furthermore, we intend to include other execution environments as for the offered parallel frameworks, eg, FastFlow, SkePU, and OpenCL SYCL. An ultimate goal is to incorporate scheduling techniques able to map task threads to CPU cores and GPUs and manage data transfers between host and device.

ACKNOWLEDGMENTS

This work has been partially supported by the EU project ICT 644235 “REPHRASE: REfactoring Parallel Heterogeneous Resource-aware Applications” and the Spanish “Ministerio de Economía y Competitividad” under the grant TIN2016-79673-P “Towards Unification of HPC and Big Data Paradigms.”

REFERENCES

1. Amarasinghe S, Hall M, Lethin R, et al. ASCR Programming Challenges for Exascale Computing [Technical Report], U.S. DOE Office of Science (SC); 2011.
2. Enmyren J, Kessler CW. 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, HLPP '10 ACM; 2010; New York, NY, USA:5-14. https://doi.org/10.1145/1863482.1863487.
3. Aldinucci M, Danelutto M, Kilpatrick P, Torquati M. FastFlow: high-level and efficient streaming on multi-core. In: Pllana S, ed. *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing. Hoboken, New Jersey, USA: John Wiley & Sons, Inc.; 2012:13.
4. Reinders J. *Intel Threading Building Blocks - Outfitting C++ for Multi-Core Processor Parallelism*. Sebastopol, California: O'Reilly Media; 2007.
5. Danelutto M, Matteis TD, Mencagli G, Torquati M. Parallelizing high-frequency trading applications by using c++11 attributes. Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA - Volume 03, TRUSTCOM-BIGDATASE-ISPA '15. Washington, DC, USA: IEEE Computer Society; 2015:140-147.

6. Kegel P, Schellmann M, Gorlatch S. Comparing programming models for medical imaging on multi-core systems. *Concurrency Computat: Pract Exper*. 2011;23(10):1051-1065.
7. Michailidis PD, Margaritis KG. Scientific computations on multi-core systems using different programming frameworks. *Appl Numer Math*. 2016;104:62-80.
8. González-Vélez H, Leyton M. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Softw Pract Exper*. Nov 2010;40(12):1135-1160.
9. del Rio Astorga D, Dolz MF, Sanchez LM, Blas JG, García JD. C++ generic parallel pattern interface for stream processing. In: Proceedings of the 16th International Conference Algorithms and Architectures for Parallel Processing (ICA3PP 2016). Cham, Switzerland: Springer International Publishing; 2016:74-87.
10. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA USA: Addison-wesley Longman Publishing Co., Inc.; 1995.
11. Rabhi FA, Gorlatch S, eds. *Patterns and Skeletons for Parallel and Distributed Computing*. London, UK, UK: Springer-Verlag; 2003.
12. McCool M, Reinders J, Robison A. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st ed. San Francisco, CA USA: Morgan Kaufmann Publishers Inc.; 2012.
13. Newburn CJ, So B, Liu Z, et al. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In: 2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Chamonix, France; 2011:224-235.
14. Kist D, Pinto B, Bazo R, Bois ARD, Cavalheiro GGH. Kanga: a skeleton-based generic interface for parallel programming. In: 2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW). Florianópolis, SC, Brazil; 2015:68-72. https://doi.org/10.1109/SBAC-PADW.2015.16.
15. NVIDIA Corporation. Thrust. https://thrust.github.io/. Accessed November 15, 2016.
16. Khronos OpenCL Working Group. SYCL: C++ Single-source Heterogeneous Programming For openCL. https://www.khronos.org/sycl. Accessed May 2015.
17. ISO/IEC. Programming languages – technical specification for c++ extensions for parallelism July 2015: ISO/IEC TS; 2015.
18. 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, PGAS '14 ACM; 2014; New York, NY, USA:6:1-6:11.
19. Beard JC, Li P, Chamberlain RD. 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, PMAM '15 ACM; 2015; New York, NY, USA:96-105. https://doi.org/10.1145/2712386.2712400.
20. Mattson T, Sanders B, Massingill B. *Patterns for Parallel Programming*. 1st ed. Reading, Massachusetts: Addison-Wesley Professional; 2004.
21. Aldinucci M, Pezzi GP, Drocco M, Spampinato C, Torquati M. Parallel visual data restoration on multi-gpgpus using stencil-reduce pattern. *Int J High Perform Comput Appl*. 2015;29(4):461-472.

How to cite this article: del Rio Astorga D, Dolz MF, Fernández J, García JD. A generic parallel pattern interface for stream and data processing. *Concurrency Computat: Pract Exper*. 2017;29:e4175. https://doi.org/10.1002/cpe.4175