The Loop-of-Stencil-Reduce paradigm

The Loop-of-Stencil-Reduce paradigm

M. Aldinucci*, M. Danelutto†, M. Drocco*, P. Kilpatrick‡, G. Peretti Pezzi§ and M. Torquati†
*Computer Science Department, University of Turin, Italy.
†Computer Science Department, University of Pisa, Italy.
‡Computer Science Department, Queen's University Belfast, UK.
§Swiss National Supercomputing Centre, Switzerland.

***Abstract*—In this paper we advocate the *Loop-of-stencil-reduce* pattern as a way to simplify the parallel programming of heterogeneous platforms (multicore+GPUs). *Loop-of-Stencil-reduce* is general enough to subsume *map*, *reduce*, *map-reduce*, *stencil*, *stencil-reduce*, and, crucially, their usage in a loop. It transparently targets (by using OpenCL) combinations of CPU cores and GPUs, and it makes it possible to simplify the deployment of a single stencil computation kernel on different GPUs. The paper discusses the implementation of *Loop-of-stencil-reduce* within the FastFlow parallel framework, considering a simple iterative data-parallel application as running example (Game of Life) and a highly effective parallel filter for visual data restoration to assess performance. Thanks to the high-level design of the *Loop-of-stencil-reduce*, it was possible to run the filter seamlessly on a multicore machine, on multi-GPUs, and on both.**

***Keywords*—skeletons, fastflow, parallel patterns, multi-core, OpenCL, GPUs, heterogeneous platforms**

I. Introduction

Since their appearance in the High-Performance Computing arena, GPUs have been widely perceived as data-parallel computing machines. This belief stems from their execution model, which prohibits any assumption about work-items/threads execution order (or interleaving) in a kernel execution. This in turn requires the avoidance of true data dependencies among different parallel activities. It quickly became clear that the best approach to programming GPUs is to *"think data-parallel"* by way of *"data-parallel building blocks"* [1], i.e. data parallel skeletons [2]. For this reason, GPUs kernels are typically designed to employ the *map-reduce* parallel paradigm, where the *reduce* is realised as a sequence of partial (workgroup-level) GPU-side reduces, followed by a global host-side reduce. Thanks to GPUs' globally shared memory, a similar pattern can be used to map computation over stencils (i.e. data overlays with non-empty intersection), provided they are accessed in read-only fashion to enforce deterministic behaviour. Often, this kind of kernel is called in host code in a loop body (e.g. up to a convergence criterion).

In this work we introduce the *Loop-of-stencil-reduce* pattern, as an abstraction of this general parallelism exploitation pattern in heterogeneous platforms. Specifically, *Loop-of-stencil-reduce* is designed as a FastFlow [3] pattern, which can be nested in other stream parallel patterns, such as *farm* and *pipeline*, and implemented in C++ and OpenCL.

We advocate *Loop-of-stencil-reduce* as a comprehensive meta-pattern for programming of GPUs because it is sufficiently general to subsume *map*, *reduce*, *map-reduce*, *stencil*, *stencil-reduce*, and, crucially, their usage in a loop, i.e. implementing the previously mentioned "data-parallel building blocks". Also, as discussed in Sec. III, it is more expressive than previously mentioned patterns.

Moreover, it simplifies GPU exploitation. In particular, it takes care of device detection, device memory allocation, host-to-device (H2D) and device-to-host (D2H) memory copy and synchronisation, reduce algorithm implementation, management of persistent global memory in the device across successive iterations, and enforces data race avoidance due to stencil data access in iterative computations. It can transparently exploit multiple CPUs or GPUs (sharing host memory) or a mix of them. Also, the same host code can exploit both a CUDA and OpenCL implementation (whereas the kernel functions should match the selected language).

While this paper builds on previous results [4], it advances them in several directions:

1) The *Loop-of-stencil-reduce* pattern is an evolution of the *stencil-reduce* pattern [4]. Specifically, *Loop-of-stencil-reduce* has been refined to explicitly include the iterative behaviour and the optimisations enabled by the knowledge of iterative behaviour. They are related to the GPU persistent global memory usage, stencil and reduce pipelining.
2) The *Loop-of-stencil-reduce* pattern has been uniformly implemented in OpenCL and CUDA, whereas *stencil-reduce* was implemented only in CUDA and using CUDA-specific features not supported in OpenCL, such as Unified Memory. Its implementation in OpenCL is particularly important in the perspective of using the pattern in heterogeneous platforms including different hardware accelerators, such as FPGAs and DSPs.
3) Support for the exploitation of iterative, locally-synchronous computations (by way of halo-swap) across multiple GPUs has been introduced, whereas in previous works usage of multiple GPUs is possible only on independent kernel instances.

The structure of the paper is as follows: in the next section related work is presented; and a recap of the FastFlow programming framework is given. Section III introduces the *Loop-of-stencil-reduce* design principles, its API, and its implementation within the FastFlow framework. Experimental results are discussed in Sec. IV: the performances of different deployments of an effective but computationally-demanding video restoration application [5] are presented. Section V presents concluding remarks.

II. Related Work

Algorithmic skeletons have been around since the '90s as an effective means of parallel application development.

An algorithmic skeleton is a general-purpose, parametric parallelism-exploitation pattern [6].

Most skeletal frameworks (or indeed, high-level parallel programming libraries) eventually exploit either low-level tools such as NVidia CUDA or OpenCL to target hardware accelerators. CUDA is known to be more compliant to C++ and often more efficient than OpenCL. On the other hand, OpenCL is implemented by different hardware vendors such as Intel, AMD, and NVIDIA, making it highly portable and allowing the code written in OpenCL to be run on different graphical accelerators.

OpenMP is a popular thread-based framework for multicore architectures mostly targeting data parallel programming. OpenMP supports, by way of language pragmas, the low-effort parallelisation of sequential programs; however, these pragmas are mainly designed to exploit loop-level data parallelism (e.g. *do_independent*). OpenMP does not natively support either *farm* or *Divide&Conquer* patterns, even though they can be implemented by using tasking features. Intel *Threading Building Blocks* (TBB) [7] is a C++ template library which provides easy development of concurrent programs by exposing (simple) skeletons and parallel data structures used to define tasks of computations.

Also, several programming frameworks based on algorithmic skeletons have been recently extended to target heterogeneous architectures. In Muesli [8] the programmer must explicitly indicate whether GPUs are to be used for data parallel skeletons. StarPU [9] is focused on handling accelerators such as GPUs. Graph tasks are scheduled by its run-time support on both the CPU and various accelerators, provided the programmer has given a task implementation for each architecture.

Among related works, the SkePU programming framework is the most similar to the present work [2]. It provides programmers with GPU implementations of several data parallel skeletons (e.g. Map, Reduce, MapOverlap, MapArray) and relies on StarPU for the execution of stream parallel skeletons (pipe and farm). The FastFlow stencil operation we introduce in this paper behaves similarly to the SkePU overlay skeleton (in some ways it was inspired by it). The main difference is that the SkePU overlay skeleton relies on a SkePU-specific data type and, to the best of our knowledge, it is not specifically optimised for being used inside a sequential loop. Another similar work in terms of programming multi-GPU systems is SkelCL, a high-level skeleton library built on top of OpenCL code which uses container data types to automatically optimize data movement across GPUs [10].

Also, the FastFlow parallel programming environment has recently been extended to support GPUs via CUDA [4] and OpenCL (as described in the present work). FastFlow CPU implementations of patterns are realised via non-blocking graphs of threads connected by way of lock-free channels [11], while the GPU implementation is realised by way of the OpenCL bindings and offloading techniques. Also, different patterns can be mapped onto different sets of cores or accelerators and so, in principle, can use the full available power of the heterogeneous platform.

III. The *Loop-of-stencil-reduce* meta-pattern in FastFlow

In the following the semantics and the FastFlow implementation of *Loop-of-stencil-reduce* are introduced. The well-known Conway's Game-of-life is used as simple but paradigmatic example of locally synchronous data-parallel applications (running on multiple devices).

A. Semantics of the Loop-of-stencil-reduce meta-pattern

Let **map** $f[a_0, a_1, \dots a_n] = [f(a_0), f(a_1), \dots f(a_{n-1})]$ and **reduce** $\oplus\ [a_0, a_1, \dots a_{n-1}] = a_0 \oplus a_1 \oplus \dots a_{n-1}$, where $f : T \to T$ is the *elemental function*, $\oplus : T \times T \to T$ the *combinator* (i.e. a binary associative operator) and $a = [a_0, a_1, \dots a_{n-1}] \in T^n$ an array of atomic elements. Let **stencil** $g\ k\ a' = [g(S_0), g(S_1), \dots g(S_{n-1})]$, where $S_i = [a'_{i-k}, \dots a'_{i+k}]$ is the i-th *neighbourhood*, and a' is the infinite extension of a (i.e. $\perp$ where a is not defined). In this work we consider a more general formulation of the stencil pattern, namelly: **stencil** $g\ k\ a' = [g(a', 0), g(a', 1), \dots g(a', n)]$, which allows the function g to access an arbitrary neighbourhoods of elements from the input array. Notice that in both formulations some care must be taken to deal with undefined values $a'_i = \perp$.

We remark that, under a functional perspective, map and stencil patterns are very similar, the only difference being the fact that the stencil elemental function takes as input a set of atomic elements rather than a single atomic element. Nevertheless, from a computational perspective the difference is substantial, since the semantics of the map leads to *in-place* implementation, which is in general impossible for stencil. These parallel paradigms have been proposed as patterns both for multicore and distributed platforms, GPUs, and heterogeneous platforms [12], [2]. They are well-known examples of data-parallel patterns, since the elemental function of a map/stencil can be applied to each input element independently of the others, and also applications of the combinator to different pairs in the reduction tree of a stencil can be done independently, thus naturally inducing a parallel implementation.

The basic building block of *Loop-of-stencil-reduce* is the stencil-reduce pattern [4], which applies a reduce pattern to the result of a stencil application (i.e. functional composition). The stencil-reduce computation is iteratively applied, using the output of the stencil at the i-th iteration as the input of the $(i+1)$-th stencil-reduce iteration. Moreover, it uses the output of the reduce computation at the i-th iteration, together with the iteration number, as input of the *iteration condition*, which decides whether to proceed to iteration $i+1$ or stop the computation. We remark that, under a pure functional perspective, the *Loop-of-stencil-reduce* can be simply regarded as a chain of functional compositions. A 2-D formulation follows directly by replacing arrays with matrices. Since the stencil pattern is a generalisation of map, it follows that any combination of the aforementioned patterns (e.g. map-reduce, Loop-of-map-reduce etc.) is subsumed by *Loop-of-stencil-reduce*.

B. The Game of Life example

We use Conway's Game of Life cellular automaton [13] as a running example in order to show the expressiveness of*Loop-*

```
1  state [0] <- NxN random binary matrix //seed
2  gen <- 0
3  do {
4    in <- state[gen]
5    out <- state[gen+1]
6    for i in 1 to N
7      for j in 1 to N
8        //count alive neighbours
9        n_alive <- 0
10       if(i>0 & j>0) //top-left
11         n_alive <- n_alive + in[i-1,j-1]
12       ...
13       if(i<N-1 && j<N-1) //bootom-right
14         n_alive <- n_alive + in[i+1, j+1]
15       //apply rules
16       out[i,j] <- n_alive = 3 | (in[i,j] & n_alive = 2)
17   gen <- gen + 1
18   //count total alive cells
19    total_alive <- sum(out)
20 } while( total_alive > 0 && gen < G)
```

Figure 1 – *Game of Life pseudocode.*

```
1  function gol_kernel2D (in, i, j, N)
2    //count alive neighbours
3    n_alive <- 0
4    if(i>0 & j>0) //top-left
5      n_alive <- n_alive + in[i-1,j-1]
6    ...
7    if(i<N-1 && j<N-1) //bottom-right
8      n_alive <- n_alive + in[i+1, j+1]
9    //apply rules
10   return n_alive = 3 | (in[i,j] & n_alive = 2)
11
12 function reduce_kernel (x, y)
13   return x + y
14
15 function iter_condition (reduce_var, iteration )
16   return (reduce_var > 0 && iteration < G)
```

Figure 2 – *Formulation of Game of Life in terms of* Loop-of-stencil-reduce.

of-stencil-reduce. The universe of the game is a matrix of cells (for simplicity, we consider a finite non-toroidal world), where each cell can be in two states: alive or dead. The first generation is created by applying a set of transition rules to every cell in the initial matrix. The process is iterated for generating generation $i+1$ from generation i, until either every cell is dead or an upper bound G on the number of iterations has been reached. During a transition from one generation to the next, the events on each cell occur simultaneously in an atomic step of time (tick), and so each generation is a pure function of the preceding one. At each tick, each cell interacts with its eight neighbours and might turn into a live or dead cell depending on the number of live cells in its neighbourhood.

Fig. 1 presents the pseudocode of a sequential algorithm for Game of Life. The initial $N \times N$ binary matrix is randomly initialised (line 1); then a do-while loop iterates over generations, until either all cells are dead or the generation index reaches the limit G (lines 3–20); at each iteration, for each cell at position (i, j), the number of live neighbours is computed (lines 9–14); an if-clause for each neighbour avoids accessing undefined values (see III-A) by excluding out-of-borders positions (lines 10, 13); the Game of Life rule is applied to decide if the cell has to live or die at the next generation (line 16); finally, the total count of live cells is computed (line 19) for checking the iteration condition (line 20).

The building blocks of the *Loop-of-stencil-reduce* meta-pattern can be easily extracted from the above pseudo-code in order to build a *Loop-of-stencil-reduce* formulation of Game of Life, which is illustrated in Fig. 2:

- stencil elemental function (lines 1–10);
- reduce combinator (lines 7–10);
- iteration condition (lines 15–16).

C. *The FastFlow* Loop-of-stencil-reduce *API*

In FastFlow, the *Loop-of-stencil-reduce* is aimed at supporting CPU only and CPU+GPU platforms by using OpenCL (or CUDA). The FastFlow framework provides the user with constructors for building *Loop-of-stencil-reduce* instances, i.e. a combination of parametrisable building blocks:

- the OpenCL code of the elemental function of the stencil;
- the C++ and OpenCL codes of the combinator function;
- the C++ code of the iteration condition.

The language for the *kernel* codes implementing the elemental function and the combinator – which constitute the business code of the application – can be device-specific or coded in a suitably specified C++ subset (e.g. REPARA C++ open specification [14]). Functions are provided that take as input the business code of a kernel function (elemental function or combinator) and translate it into a fully defined OpenCL kernel, which will be offloaded to target accelerator devices by the FastFlow runtime. Note that, from our definition of elemental function (Sec. III-A), it follows that the *Loop-of-stencil-reduce* programming model is data-oriented rather than thread-oriented, since indexes refer to the input elements rather than the work-items (i.e. threads) space, which is in turn the native programming model in OpenCL.

In order to build a *Loop-of-stencil-reduce* instance, the user also has to specify two additional parameters controlling parallelism: 1) the number of accelerator devices to be used (e.g. number of GPUs in a multi-GPU platform) and, 2) the maximum size of the neighbourhood accessed by the elemental function when called on each element of the input. Note that the second parameter could be determined by a static analysis on the kernel code in most cases of interest, i.e. ones exhibiting a static stencil (e.g. Game of Life) or dynamic stencil with reasonable static bounds (e.g. Adaptive Median Filter, [5]). Once built, a *Loop-of-stencil-reduce* instance can process *tasks* by applying the iterative computation described in Sec. III-A to the input of the task, by way of the user-defined building blocks. An instance can run either in *one-shot* (i.e. single task) or *streaming* (i.e. multi-task) mode. In streaming mode, independent tasks can be offloaded to different GPUs, thus exploiting inter-task parallelism.

Moreover, intra-task parallelism can be employed by offloading a single task to a *Loop-of-stencil-reduce* instance deployed onto different GPUs. Although this poses some challenges at the FastFlow implementation level (see Sec. III-E), at the API level it requires almost negligible refactoring of user code. That is, when defining the OpenCL code of the elemental function, the user is provided with local indexes over the index space of the device-local sub-input – to be used when accessing the input – along with global indexes over the index space of the whole input – to be used to e.g. check the absolute position with respect to input size.

```
1  std :: string   stencilf  = ff_stencilKernel2D_OCL(
2        "unsigned char", "in", //element type and input
3        "N", "M", //rows and columns
4        "i", "j", "i_", "j_", //row-column global and local indexes
5          std :: string ("") +
6          /* begin OpenCL code */
7          "unsigned char n_alive = 0;\n"
8          + "n_alive += i>0 && j>0 ? in[i_-1][j_-1] : 0;\n"
9           ...
10         + "n_alive += i<N-1 && j<M-1 ? in[i_+1][j_+1] : 0;\n"
11         + "return (n_alive == 3 || (in[i_][j_] && n_alive == 2));"
12         /* end OpenCL code */);
13
14 std :: string  reducef = ff_reduceKernel_OCL(
15     "unsigned char", "x", "y", "return x + y;");
16
17 ff :: ff_stencilReduceLoop2DOCL<golTask> golSRL(
18   stencilf , reducef , 0, iterf , /* building blocks */
19   N, N, /* matrix size */
20   NACC, 3, 3);
```

Figure 3 – *Implementation of Game of Life on top of the* Loop-of-stencil-reduce *API in FastFlow.*

Fig. 3 illustrates a Game of Life implementation on top of the *Loop-of-stencil-reduce* API in FastFlow. Source-to-source functions are used to generate OpenCL kernels for both stencil elemental function (lines 1–12) and reduce combinator (lines 14–15). The source codes – OpenCL versions of the pseudocode in Fig. 2 – are wrapped into fully defined, efficient OpenCL kernels. The user, in order to enable exploitation of intra-task parallelism, has to use local indexes i_ and j_ to access elements of the input matrix. C++ codes for iteration condition and reduce combinator are not reported, as they are trivial single-line C++ lambdas. The constructor (lines 17–19) builds a *Loop-of-stencil-reduce* instance by taking the user-parametrised building blocks as input, plus the identity element for the reduce combinator (0 for the sum) and the parameters for controlling intra-task parallel behaviour, namely the number of devices to be used over a single-task (NACC) and the 2D maximum sizes of the neighbourhood accessed by the elemental function (Game of Life is based on 3-by-3 neighbourhoods). Finally, the constructor is parametrised with a template type golTask which serves as an interface for basic input-output between the application code and the *Loop-of-stencil-reduce* instance.

FastFlow does not provide any automatic facility to convert C++ code into OpenCL code. It does, however, facilitate this task via a number of features including:

- Integration of the same pattern-based parallel programming model for both CPUs and GPUs. Parallel activities running on CPUs can be either coded in C++ or OpenCL.
- Setup of the OpenCL environment.
- Simplified data feeding to both software accelerators and hardware accelerators (with asynchronous H2D and D2H data movements).
- Orchestration of parallel activities and synchronisations within kernel code (e.g. reduce tree), synchronisations among kernels (e.g. stencil and reduce in a loop), management of data copies (e.g. halo-swap buffers management).
- Transparent usage of multiple GPUs on the same box (sharing the host memory).

```
1 while cond
2   before (...)  // On host, iteration initialisation , possibly in parallel
        on CPU cores
3   prepare (...) // On device, swap I/O buffers , set kernel args, d2d-sync
        overlays
4   stencil <SUM_kernel,MF_kernel> (input, env) // On GPU, stencil and
        partial reduce
5   reduce op data  // On host, final reduction
6   after (...)  // On host, iteration finalisation , possibly in parallel on
        CPU cores
7 read(output) //d2h-copy output
```

Figure 4 – Loop-of-stencil-reduce *pattern general schema.*

D. Loop-of-stencil-reduce *expressiveness*

In the shared-memory model, *Loop-of-stencil-reduce* exhibits an expressiveness similar to that of the well-known map-reduce paradigm, where the map is apply-to-all to a set of elements (list, array) and computations of different elements are independent. In fact there exists a quite straightforward way to express one in terms of the other. The *Loop-of-stencil-reduce* pattern can be trivially configured to behave as map-reduce (i.e. the set of neighbours per element is the element itself). Also, there are several methods to exploit a stencil with a map on a GPU, such as using the map pattern over overlays or exploiting shared global memory. This requires a specific data structure to manage overlays (sharing or copies). This approach is exploited in the SkePU framework via *MapOverlap* and *MapArray*, respectively [2].

We advocate *Loop-of-stencil-reduce* adoption because it explicitly exposes data dependencies at the pattern declaration level (see Fig. 3, line 20). This naturally describes a wide class of data parallel applications. Also, making the stencil explicit at the API level enables the kernel developer to reason about optimisations related to local memory, memory alignment, and static optimisation of halo buffers in the distributed memory space of multiple GPUs.

E. The FastFlow implementation

The iterative nature of the *Loop-of-stencil-reduce* computation presents challenges for the management of the GPU's global memory across multiple iterations, i.e. across different kernel invocations.

The general schema of the *Loop-of-stencil-reduce* pattern is described in Fig. 4. Its runtime is tailored to efficient loop-fashion execution. When a task is submitted to be executed by the devices onto which the pattern is deployed, the runtime takes care of allocating on-device global memory buffers and filling them with input data via H2D copies. The naive approach for supporting iterative computations on a hardware accelerator device equipped with some global memory (e.g. GPU) would consist in putting a global synchronisation barrier after each iteration of the stencil, reading the result of the stencil back from the device buffer (full size D2H copy), copying back the output to the device input buffer (full size H2D copy) and proceeding to the next iteration. FastFlow in turn employs *device memory persistence* on the GPU across multiple kernel invocations, by just swapping on-device buffers. In the case of multi-device intra-task parallelism (Sec. III-C), small device-to-device copies are required after

each iteration, in order to keep halo borders aligned, since no device-to-device copy mechanism is available (as of OpenCL 2.0 specification, device-to-device transfers). Global memory persistence is quite common in iterative applications because it drastically reduces the need for H2D and D2H copies, which can severely limit the speedup. This also motivates the explicit inclusion of the iterative behaviour in the *Loop-of-stencil-reduce* pattern design which is one of the differences with respect to solutions adopted in other frameworks, such as SkePU [2].

As a further optimisation, FastFlow exploits OpenCL events to keep *Loop-of-stencil-reduce* computation as asynchronous as possible. No dependencies exist between stencil and reduce computations at different iterations. Put another way, stencil and reduce computations can be pipelined (i.e. stencil at iteration $i+1$ can run in parallel with reduce at iteration i). Moreover, in the case of multi-GPU intra-task parallelism, sub-tasks running on different GPUs at the same iteration are independent of each other, and so can run in parallel. By exploiting the OpenCL events API, an almost arbitrary graph of task dependencies can be implemented, thus fully exploiting all the available parallelism among operations composing a *Loop-of-stencil-reduce* computation.

We remark that providing the user with the low-level, platform-specific optimisation mentioned above, is one of the key features of the skeleton-based parallel programming approach.

IV. Experimental Evaluation

Here we present a preliminary assessment of the *Loop-of-stencil-reduce* FastFlow implementation on top of OpenCL. For this two applications are used: the Game of Life application, described in Sec. III-B; and the two-phase video restoration algorithm. For more details on the video restoration algorithm we refer to [5], [4].

All experiments were conducted on an Intel workstation with 2 eight-core double-context (2-way hyper-threading) Xeon E5-2660 @2.2GHz, 20MB L3 shared cache, 256K L2, and 64 GBytes of main memory (also equipped with two NVidia Tesla M2090 GPUs) with Linux x86_64.

a) Game of Life: Table I reports execution times of different deployments of the Game of Life application on 1) a CPU deployment with multiple threads running on the cores of a multi-core CPU and relying on the OpenCL runtime for exploiting parallelism; 2) a 1xGPU deployment running on a single M2090 NVidia GPU; 3) a 2xGPU deployment exploiting intra-task parallelism (as discussed in Sec. III-E) over two M2090 devices. First, performance usually benefits from offloading data-parallel computations onto GPU devices, as demonstrated by the fact that in all cases execution times on the GPU are faster than the respective execution times on CPU. The main factor limiting the GPU-vs-CPU speedup is the ratio of the time spent in H2D and D2H memory transfers over the effective computing time. If few iterations and/or small matrices are considered, then the overhead due to memory transfer becomes relevant and limits the impact of parallelism, in accordance with Amdahl's law. The same considerations apply to the impact of intra-task parallelism (2xGPU vs 1xGPU). The benefit of exploiting two boards are limited if the population matrix is too small, since the amount of memory to be transferred D2D after each iteration in general scales up with a fraction of the input size. In particular, for the Game of Life application, the amount of memory to be transferred is $O(N)$, thus it scales up with the square root of the input size $N \times N$. When the overhead becomes almost negligible (e.g. last row of Table I), intra-task parallelism provides almost ideal speedup.

N	CPU (ms)	1xGPU (ms)	2xGPUs (ms)
10 iterations			
1024	584.57	103.37	166.36
4096	1153.42	201.80	208.46
16384	9716.16	1599.54	1358.74
100 iterations			
1024	842.10	162.97	195.61
4096	4516.16	955.80	626.84
16384	90665.99	13842.12	7121.55
1000 iterations			
1024	4390.96	694.86	569.50
4096	39598.45	8653.62	4546.43
16384	1419309.97	135491.15	68224.51

Table I – *Execution times of different deployments of the Game of Life application. Parameters of the reported experiments are the number of rows of the population matrix (N) and the number of* Loop-of-stencil-reduce *iterations, which was fixed to a constant for each run.*

%-noise	CPU	1xGPU	2xGPUs intra-frame	2xGPUs
Throughput (fps) of video stream restoration				
10	1.01	5.37	8.26	8.30
50	0.30	3.20	5.39	5.44
90	0.25	2.74	4.73	4.76
Execution time (ms) of single image restoration				
10		120.32	76.89	
50		223.17	135.22	
90		264.01	157.83	

Table II – *(upper) Performance of different deployments of the restore stage over a VGA-resolution (640×480) video stream, under different levels of noise affecting pixels. (lower) Performance of single-GPU and multi-GPU restore stages on a high-resolution (16384×16384) image*

b) Two-phase video restoration: Two kinds of experiment are reported:

1) Performance over a video stream of different deployments (i.e. different parallelisation schemas) of the restore stage.
2) Performance on a single image of both single-device and multi-device configuration of the *Loop-of-stencil-reduce*.

Table II shows the observed results. The upper part reports the throughput (i.e. frames per second) obtained by running different deployments of the restore stage over a video stream under different noise-level conditions, which in turn require different numbers of iterations for convergence. The "CPU" deployment is the baseline: each frame is passed through a *Loop-of-stencil-reduce* OpenCL version of the filter, deployed

onto the (cores of the) CPU. Defining a single video frame as a task, this configuration exploits intra-task data parallelism on each frame. The baseline is compared against different GPU deployments of the *Loop-of-stencil-reduce*. The "1 GPU" version exploits the same intra-task parallelism as the baseline version but runs on the GPU. The "2 GPUs intra-task" version exploits intra-task data parallelism by splitting single frames on two GPUs, and finally the "2 GPUs" version exploits the "1 GPU" version on successive (independent) frames of the video stream, each offloaded to one of the two GPUs (by way of a FastFlow farm pattern).

The performance ratio among different versions is consistent with a hand-tuned development "1 GPU" [5]. For applications of this kind, the GPU deployment is not surprisingly several times faster. The deployment on 2 GPUs exhibits 65% more throughput with respect to the single GPU version. Also, the 2 GPUs version on the same video frame exhibits almost the same performance as the 2 GPUs version working on independent kernels, suggesting that the *Loop-of-stencil-reduce* succeeds in keeping the halo swap overhead quite limited.

The lower part of Table II reports the execution time of the filter when applied to a single large image. Here there is no opportunity to exploit parallelism among different frames. In this case, using a multi-device deployment of the OpenCL *Loop-of-stencil-reduce* restore stage can lead to the full exploitation of the aggregated computational power of multiple GPUs, as shown by the almost linear speedup observed.

V. Conclusions and Future Works

In this work we have presented *Loop-of-stencil-reduce*, a parallel pattern specifically targeting high-level programming for heterogeneous platforms. The *Loop-of-stencil-reduce* pattern abstracts a common data parallel programming paradigm, which is general enough to subsume several popular patterns such as map, reduce, map-reduce. It significantly simplifies the development of code targeting both multicore and GPUs by transparently managing device detection, device memory allocation, H2D/D2H memory copy and synchronisation, reduce algorithm implementation, management of persistent global memory in the device across successive iterations, and data race avoidance. The same code using *Loop-of-stencil-reduce* can be deployed on multiple GPUs and on combinations of CPU cores and GPUs. It should be noticed, however, that this latter deployment requires a careful (and not always possible) planning of load to be distributed to GPU cores and GPUs due to their difference in performance [15].

The *Loop-of-stencil-reduce* pattern has been tested on a real-world application, i.e. an image restoration application, which is typically too slow to be actually usable when implemented on a single CPU or even on a 32-core platform. Also, the application requires access to the image along three successive filtering iterations to determine the convergence of the process, thus needing a quite complex design with large temporary data sets that should be moved across different memories as little as possible. The presented design based on FastFlow *Loop-of-stencil-reduce* makes it possible to easily implement the application with comparable performance to hand-optimised OpenCL code.

Despite the fact that currently the *Loop-of-stencil-reduce* pattern is provided to programmers by way of C-style macros, we have already planned to substantially improve the embedding of the *Loop-of-stencil-reduce* pattern into the C++ language by way of C++ demacrofication process [16] and/or the C++11 attributes mechanism. This process is already ongoing within the REPARA project.

Acknowledgment

This work has been supported by the EU FP7 REPARA project (no. 609666) and by the NVidia GPU Research Center.

References

[1] J. Owens, SuperComputing 07, High Performance Computing with CUDA tutorial, 2007.

[2] J. Enmyren and C. W. Kessler, "SkePU: a multi-backend skeleton programming library for multi-gpu systems," in *Proc. of the fourth Intl. workshop on High-level parallel programming and applications*, ser. HLPP '10. New York, NY, USA: ACM, 2010, pp. 5–14.

[3] M. Danelutto and M. Torquati, "Structured parallel programming with "core" fastflow," in *Central European Functional Programming School*, ser. LNCS. Springer, 2015, vol. 8606, pp. 29–75.

[4] M. Aldinucci, G. Peretti Pezzi, M. Drocco, C. Spampinato, and M. Torquati, "Parallel visual data restoration on multi-GPGPUs using stencil-reduce pattern," *International Journal of High Performance Computing Application*, 2015.

[5] M. Aldinucci, C. Spampinato, M. Drocco, M. Torquati, and S. Palazzo, "A parallel edge preserving algorithm for salt and pepper image denoising," in *Proc of 2nd Intl. Conference on Image Processing Theory Tools and Applications (IPTA)*. IEEE, 2012, pp. 97–102.

[6] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computations*, ser. Research Monographs in Par. and Distrib. Computing. Pitman, 1989.

[7] Intel Corp., *Threading Building Blocks*, 2014. [Online]. Available: http://www.threadingbuildingblocks.org/

[8] S. Ernsting and H. Kuchen, "Data parallel skeletons for gpu clusters and multi-gpu systems," in *Proc. of PARCO 2011*. IOS Press, 2011.

[9] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[10] S. Breuer, M. Steuwer, and S. Gorlatch, "Extend ing theSkelCL Skeleton Library for Stencil Computations on Multi-GPU Systems," in *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*, A. Größlinger and H. Köstler, Eds., Vienna, Austria, Jan. 2014, pp. 15–21.

[11] 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, S. Pllana and F. Xhafa, Eds. Wiley, 2015, ch. 13.

[12] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers," *Software: Practice and Experience*, vol. 40, no. 12, 2010.

[13] M. Gardner, "Mathematical games: the fantastic combinations of John Conway's new solitaire game 'Life'," *Scientific American*, vol. 223, no. 4, pp. 120–123, oct 1970.

[14] J. D. Garcia, "REPARA C++ open specification," REPARA EU FP7 project, Tech. Rep. ICT-609666-D2.1, 2-14.

[15] J. Shen, A. L. Varbanescu, and H. Sips, "Look before you leap: Using the right hardware resources to accelerate applications," in *Proc. of 16th IEEE Intl Conference on High Performance Computing and Communications (HPCC 2014)*. IEEE, 2014, pp. 383–391.

[16] A. Kumar, A. Sutton, and B. Stroustrup, "Rejuvenating C++ programs through demacrofication," in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*. IEEE Computer Society, 2012, pp. 98–107.