Published in Proceedings of the 2015 IEEE/ACM 1st International Workshop on Software Engineering for High Performance Computing in Science (SE4HPCS 2015), https://doi.org/10.1109/SE4HPCS.2015.13

SIMPL: a Pattern Language for Writing Efficient Kernels on GPGPU

Davide Barbieri, Valeria Cardellini, Salvatore Filippone
Dipartimento di Ingegneria Civile e Ingegneria Informatica
Università di Roma "Tor Vergata", Roma, Italy
davide.barbieri@ghostshark.it, cardellini@ing.uniroma2.it, salvatore.filippone@uniroma2.it

***Abstract*—Graphics processing units (GPUs) have become an integral part of both High Performance Computing (HPC) and desktop systems. To fully exploit their potential, algorithms should be specifically designed to fit the *General Purpose computing on GPU* (GPGPU) programming paradigm and, above all, an optimized implementation should be provided. In the past, pattern languages were proven to be an effective way to communicate experience and help researchers and developers to reduce the learning curve over a particular expertise field. In this paper we describe SIMPL, a pattern language dedicated to GPGPU computing. We discuss in detail three example patterns enabling optimal performance results on various classes of applications.**

I. Introduction

Driven by the increasing demand by video game industry for better performance in realtime rendering, in recent years Graphics Processing Units (GPUs) have evolved into processors consisting in hundreds of cores with high ALU throughput and memory bandwidth. Current GPUs represent a pervasive high-performance parallel computing platform, with extensive market demand that also allows for economies of scale and therefore making them into a very cost-effective option. Implementation on GPUs of algorithms that are not strictly related to graphics has become a commonplace activity in scientific computing, as outlined by the many GPU-based machines found in the Top 500 list[1]. The GPGPU acronym stands for *General Purpose Computation Using Graphics Processing Units* and identifies the hardware/software combination allowing the use of GPUs for general purposes.

Amdahl's [1] and Gustafson's [2] laws left us a dichotomy between the task of latency reduction (that is, doing the same work in less time) and the task of throughput increase (that is, doing more work in the same time window) and they connect these tasks to the proportion of the inherently sequential part over the part that could be parallelized. Latency reduction can take advantage from cores with complex circuits for out-of-order execution, branch prediction, and large cache memories. These resources require a large amount of the chip "real estate" and for this reason complex processors tend to have relatively few cores. Amdahl's law states how the speedup attained by an increasing number of processors tends quickly to a constant on problems with a large inherent sequential percentage; this seems to suggest that a CPU with few, complex cores is better suited for this use. On the other hand, the task of throughput increase can be accelerated by an architecture with a very high number of cores. However, to fit on the same chip, these cores should be heavily simplified; an effective way to do this is to pack multiple ALUs in units working in a SIMD[2] fashion. This is the approach taken by current GPU platforms. With contemporary GPU architectures, instruction stalls are hidden by concurrency of a large number of active threads scheduled on the same multiprocessor, without the need of large caches and out-of-order execution. Moreover, the overall energy consumption rate is drastically reduced. These observations suggest that future architectures will likely continue to provide different architectural modules to execute either latency-aware or throughput-aware tasks, and presumably we will see many efforts aimed at providing integrated solutions comprising both capabilities.

Today we find octa-core CPUs and GPUs with hundred of cores even inside commodity mobile devices; nevertheless the learning curve for parallel programming is still quite steep. Trying to figure out the main difficulties arising from parallel programming we identify the following issues:

1) It is more natural to think about sequential steps than orchestrating multiple flows;
2) Parallelization opportunities are often far from obvious, especially when a solution depends on inherently sequential subtasks;
3) Specific data structures are needed to favor good memory access patterns and data reuse;
4) Modern compilers are quite efficient at optimizing sequential code, while parallel programs often require subtle algorithmic changes beyond the capabilities of current compilation techniques;
5) Even if the problem is suitable to achieve good scalability, many related programming tasks are quite effort intensive and error prone;
6) Multiple threads or processes that concurrently work on the same problem need to synchronize, and this gives rise to many challenging issues, including race conditions, deadlocks, and consistency errors;
7) Debugging is harder because faults show up apparently at random and it is very hard to follow step by step many concurrent threads;

[1] http://www.top500.org

[2] Single Instruction Multiple Data

8) Energy efficiency adds to the already long list of challenges;
9) Hardware may fail; the fault probability increases with the size of the parallel platform, so a fault tolerance model is an absolute necessity on massively parallel platforms.

All of these complications can be managed with practice and experience, but the related training takes a long time.

In the past years, we had experiences over several application fields of GPU computing, including sparse and dense linear algebra, domain partitioning techniques, cryptography, and computational fluid-dynamics [3], [4], [5], [6], [7]. During these research activites, we faced all the difficulties listed above; as a result, we propose SIMPL, a *pattern language* to communicate knowledge about GPGPU programming solutions. As first introduced in [8], a *pattern language* defines a structured collection of design practices within a field of expertise.

Proposing a pattern language does not eliminate the need for the programmer to get acquainted with the underlying architecture; however, it helps the programmer organize his thinking and it reduces the required effort in the path to proficiency, for instance the novice programmer will need to concentrate on the identification of the few parameters discussed in the pattern presentation. To the best of our knowledge, this is the first pattern language exclusively dedicated to GPGPUs.

The rest of the paper is organized as follows. In Section II we trace the "pattern" terminology and discuss related work on the use of patterns in computing. We introduce our pattern language for GPU in Section III. In Sections IV, V and VI we present three case studies, the Vectorize, Enumerate and Sort and Pack patterns, respectively, and discuss the performance improvements that arise from their application. Finally, we conclude in Section VII.

II. Related Work

The definition of "Pattern language" appeared first in the completely different context of urban design, in the book [8]; the term "Design patterns" as a software development tool gained popularity after the release of [9] by the so-called "Gang of Four", addressing solutions to common object-oriented design problems. Another important work in this field was done by Garlan and Shaw [10]; all these patterns helped many engineers, experienced or not, to understand implications of the object-oriented paradigm, discuss about software architecture, and quickly find established solutions to design problems.

Pattern-based software design has been extensively studied in parallel and distributed programming, see e.g. [11], [12], [13]. The pattern approach is considered a good tool to teach parallel and distributed computing by [14]; other works related to training with patterns include McCool [15] and Ortega [16].

Although the above works are useful to solve parallel issues related to a generic computing cluster, the patterns they discuss are seldom useful in the GPU context, because of the constraints that the single GPU imposes to the communication model that each thread should meet. Patterns based on coarse-grain task parallelism are inapplicable on the GPU architecture, which requires substantial data-parallelism. Moreover, many existing patterns treat data-parallelism in a fairly generic fashion to be as portable as possible, but their translation to an actual GPU code is far from straightforward. Mattson et al. in [17] add some extensions to their pattern language to include a SIMD data-parallelism pattern for devices like the GPU; while this work gives some good guidelines, it discusses only a single pattern.

Software patterns are sometimes embodied in *algorithmic skeletons*, that is, high-level models for parallel programming hiding as much as possible of the implementation details. Skeletons are often composed using *skeleton frameworks*; for a discussion see [18], [19], [20]. Skeleton framework can be a valuable choice for domain expert programmers that are not expert parallel programmers, while parallel design patterns may help programmers to acquire parallel programming expertise. Skeleton-based programming has portability problems; in fact, to the best of our knowledge, none of the skeleton frameworks were adapted to generate GPU kernels. Often algorithms don't perfectly fit inside skeletons, and in this case it is necessary to understand the patterns they are based on to deal with exceptions. Moreover, delving into the underlying platform details is often necessary to reach optimal performance.

III. SIMPL: a Pattern Language

A *pattern language* defines a structured collection of design practices within a field of expertise [8]. We propose a **data-parallel** pattern language for hardware based on the *Single-Instruction Multiple-Threads* (SIMT) paradigm, like CUDA or OpenCL. We call the language **SIMPL** (**SIM**T **P**attern **L**anguage); it is currently made by sixteen patterns divided into five categories. In the following we present the template used to define our design patterns, the taxonomy employed to categorize them and the underlying architectural model.

A. Pattern Template

Richard Gabriel [21] suggests the following definition for the field of software design:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves.

Usually, a common scheme used to present a design pattern comprises the following sections:

- *Name* of the pattern, should be easy to remember;
- *Context* illustrating the scenario in which it arises;
- *Problem* summarizing the goal of the pattern;
- *Forces* listing the main difficulties and conflicting requirements;
- *Solution* identifying a model that can be used to solve the problem;

- *Consequences* listing the main constraints that the solution imposes to the resulting design;
- *Example uses*.

Our pattern language, as opposed to other parallel pattern languages found in literature, is focused on optimization of parallel algorithms on a single architectural model, that is a SIMT machine. As such, all patterns share a common *context* which is therefore not listed explicitly. In a similar way, all of our patterns share a common set of forces affecting parallel programs on a SIMT device:

- Efficiency, saturating memory bandwidth, ALU throughput or both;
- Portability;
- Scalability;
- Reproducibility of results[3];
- Maintainability;
- Simplicity (as much as possible!);
- Avoidance of locking schemes (as much as possible).

Additional forces may appear in a given pattern discussion.

B. Taxonomy

The taxonomy used by SIMPL is based on a list of common problems, described in Section I, that arise in parallel computing. Based on these, we construct the five basic categories of our pattern language:

- Mapping Patterns: adapting a parallel algorithm to a SIMT architecture in an efficient way;
- Consistency Patterns: help designing semantically correct and high performance synchronization between threads;
- Transformation Patterns: used to transform a problem to expose data-parallelism;
- Construction Patterns: building data structures in parallel;
- Tuning Patterns: exposing parameters to fine-tune kernel performance on different architectures.

From the list in Section I we are still missing some categories that will be the subject of future work; they include *Fault Tolerance*, *Energy Efficiency* and *Debugging/Profiling*.

C. Underlying Architecture Model

Given the peculiarities and the constraints of a throughput-oriented architecture, a meaningful discussion about optimization requires some background on the programming paradigm and the hardware architecture. However, creating a pattern language focused on just one concrete architecture would be too limiting; therefore we provide a simplified and sufficiently general model of SIMT architecture and its programming paradigm, on which to base our language. We try to focus on common implementation features of GPUs that are likely to be present in future as well as current devices. Similarly, we concentrate on a subset of optimal memory access patterns to ensure portability on a wide set of GPU architectures.

The programming model follows the basic principles of the *CUDA* and *OpenCL* languages. A program (*kernel*) is invoked on the GPU (*device*) using a *remote procedute call* made by the *host*, that is, the CPU along with its RAM. In the invocation (also called *grid*), the host defines the execution configuration, that is:

- how many blocks of threads should be executed, possibly defined using multiple coordinates;
- the number of threads per block, defined possibly using multiple coordinates;

The programming environment provides constraints on the number of blocks per grid and of threads per block. Each

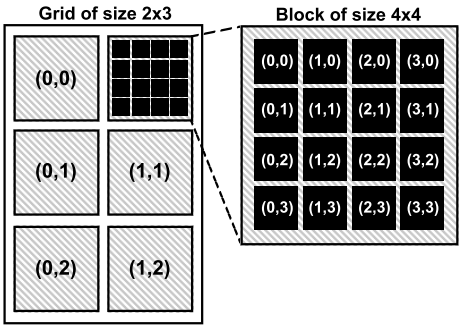

Fig. 1. A 2D grid of threads

thread obtains an identifier within the block and an identifier of its block within the grid, as shown in Figure 1. All threads share the same entry point in the kernel; their different identifiers can be used for specialization, for example, to address different offsets inside linear memory.

Figures 2 and 3 describe the underlying Single-Instruction Multiple-Threads (SIMT) architecture. As shown in Figure 2,

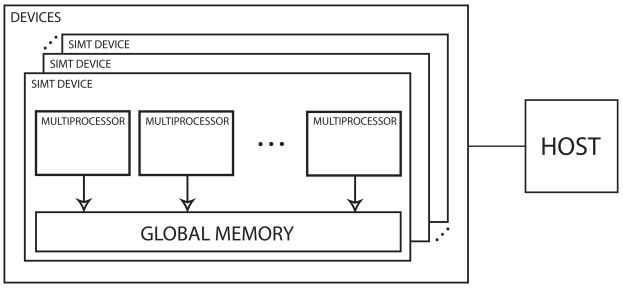

Fig. 2. SIMT model: host and devices

a single host may coexist with multiple devices. Each device is made up by an array of multiprocessors and a global memory; no cache hierarchy is considered. The global memory is divided in channels (or banks). An address $addr$ is within channel Ch if $(\frac{addr}{channelSize} \bmod numChannels) == Ch$; the size of a channel $channelSize$ is usually equal to 256 bytes. Memory requests to the same channel are enqueued and each channel provides only a fraction of the whole bandwidth; to exploit the full device bandwidth the grid should access all channels at the same time.

The host is connected to devices using a bus, often with a much smaller bandwidth than the device global memory.

[3]Exact reproducibility is not always possible, e.g. when using floating-point operations.

Each multiprocessor in Figure 3 contains a set of schedulers,

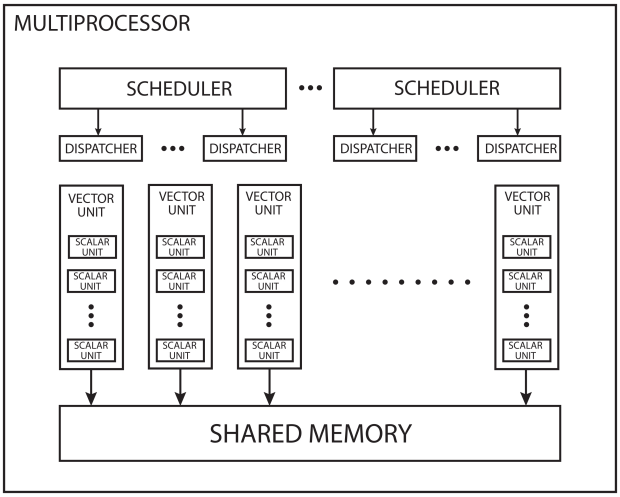

Fig. 3. SIMT model: a multi-processor

a set of vector units, a set of registers, and a shared memory. Multiprocessors execute only vector instructions; a vector instruction specifies the execution on a set of threads (called *warp*, or *wavefront*) with contiguous identifiers inside the block. The warp size is a characteristic of the architecture (currently, it is 32 for Nvidia's GPUs and 64 for AMD's GPUs). A grid is executed on a single device; each thread block is enqueued and then scheduled on a multi-processor with enough available resources (registers, shared memory and block slots) and retains all its resources until completion. A warp is issued by a *scheduler* on an available vector unit that supports that class of instructions. If threads in the same warp execute divergent code branches, the scheduler issues instructions in turn for the various control flows, and masks the threads executing each of them. If a warp scheduler has more than one *dispatcher*, multiple independent instructions from the same warp may be issued. The throughput of a class of instructions will thus depend on: the number of schedulers and dispatchers, the number of vector units supporting the instruction class, the number of internal scalar units, and the warp size. Threads belonging to the same thread block can share data using *shared memory* and can synchronize waiting on a *barrier*.

The shared memory is divided into banks; to exploit its full throughput, warp instructions that access shared memory must avoid bank conflicts. In our model, the number of banks B is equal to the warp size and each bank is four bytes wide. Accesses to global memory may be modeled in the same way. For best performance each thread with index k within the warp ($0 \leq k < warpSize$) should access the element of size D (with D equal to 4, 8 or 16 bytes) at address $D \cdot (Offset + k)$. A memory access that follows these rules is called a *coalesced access*; the cost of this access is proportional to the number of aligned blocks of size $warpSize \cdot 4$ bytes.

These access patterns work on all current GPU architectures. Some specific models may either impose additional requirements or relax some constraint. For example, old NVIDIA cards with compute capability 1.0 and 1.1 (deprecated in CUDA 6.5) require that $Offset$ is a multiple of $D \cdot (warpSize/2)$, while the AMD GCN architecture does not support coalescing for 64-bit wide reads.

Due to space limitations, we will describe in details only three patterns in Sections IV, V, and VI; a full description of all patterns currently included in SIMPL will be the subject of future publications.

IV. Vectorize Pattern

Problem

How to map a data-parallel algorithm on a SIMT architecture in a way that fully exploits its compute performances?

Forces

- To fully exploit the many-core parallelism, threads should work in lock-step; warps (32) and wavefronts (64) are examples of this vector processing feature;
- To fully exploit the high memory bandwidth of these devices, accesses to memory should occur in a coalesced fashion, that is threads executing the same vector instruction should read or write aligned and sequential linear addresses of size D. Acceptable values of D are usually 4 bytes, 8 bytes or 16 bytes.

Solution

Re-parametrize your program in such a way that the fundamental unit of data (i.e., the piece of data that a single thread computes) to be processed in each part of the code is not a single value (e.g., a float or an integer), but rather a vector of K values. Then, choose K according to a multiple of the vector size of the computing and load/store units of the target architecture (e.g., 32 for Nvidia GPUs, 64 for AMD GPUs). This naturally maps to operations in which threads with increasing identifier, belonging to the same warp/wavefront, operate in lock-step mode, accessing subsequent elements in memory.

Very often in programming, the different properties of a single domain element for a particular application are stored in a structure. When we have multiple elements to process, we have an array of structures. To apply the Vectorize pattern to the general case, we turn this array of structures into a structure of arrays, in order to pack values of the same property for different elements in sequence in the same allocation. Doing this way, it is easier to access and process the data of a vector of K near elements at once. On a GPU, this applies also if the properties are themselves small vectors (float2/float4).

The single property sometimes occupies a relatively large amount of memory space, say $P \cdot D$ (with D equal to 4 bytes, 8 bytes or 16 bytes), which could not be read using a single memory instruction. To accomodate this, it is possible to transform the array of N elements of size $P \cdot D$ into a matrix (stored column-major) of $N \times P$ elements of size D.

Since N is ensured to be a multiple of K, operations like *"load the i-th part of the property for each vector element"*

would be compliant with the memory access requirements. This is true also on multidimensional domains, since the last vector element of each row would implicitly contain the padding needed to align accesses for the subsequent rows.

Consequences

The main consequence of this pattern is that the data structures involved in the whole application should be modified to accomodate this vector nature. This implies also that host functions should be changed accordingly. If this is not feasible or convenient, the application should manage both formats and use conversion routines when host and device should exchange data. In [6] we present an object-oriented architectural model that represents a seamless integration of the GPU support for a pre-existing sparse computations library ([22], [23]).

Case Study

This is the most frequent pattern inside GPU implementations, and most likely the most frequent pattern used on each vector platform. We applied it to the design of dense and sparse linear algebra routines ([3], [24], [4], [6]) and to the simulation of interacting particles [5].

Here we provide an example of its application on a sparse linear algebra application. The multiplication of a sparse matrix by a dense vector (spMV) is a centerpiece of scientific computing applications and the CSR (Compressed Sparse Row) format is one of the most popular formats for sparse matrices. As illustrated in Figure 4, the CSR format comprises three arrays. The first two are used to store *non zero* entries (AS array) and their relative column index (JA array) and are sorted per row index. In the third array (IRP), the *i-th* element represents the starting position, in the previous two arrays, of the row *i*. Let us assume that our multiply routine

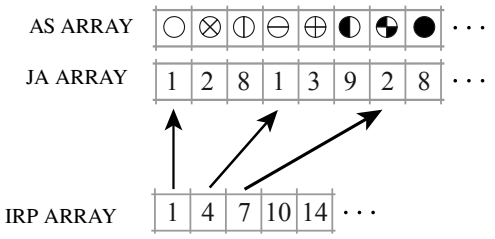

Fig. 4. CSR sparse matrix format

$y = Ax$ assigns the computation of a different resulting element of y to each thread. Thus, each thread will read a full row of non zero entries, use their column indices to identify and access which elements in x should be read, and perform the computation. We note that there is not a way, in the general case, to efficiently read both x and A. We apply here the vectorize pattern to efficiently read the matrix A. To read a row from CSR format, a thread would read two elements from the *IRP* array to know where its row begins and where it ends, then it would read the row's elements in the *AS* and *JA* arrays. To apply the vectorize pattern, we consider a *row* as the fundamental unit of data. So we modify the CSR format in such a way that a set of W rows (with W equal either to the warp size or to a multiple of it) is considered. The parts that constitute a row, in this case, are of course its elements, since each one could be read using a single read instruction. Following the pattern, we store those elements as depicted in Figure 5. Inside the memory those matrices (called *hacks* in our HLL format) are stored in column-major format, and each hack is stored sequentially. The IRP array is replaced by a new array that stores the start element of each hack instead of the single row. All

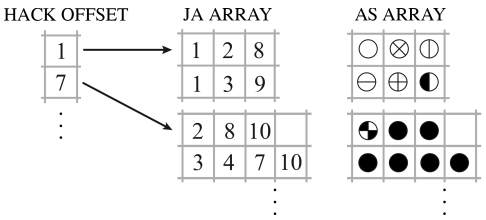

Fig. 5. HLL sparse matrix format

TABLE I
SPMV PERFORMANCE (GFLOP/S) - NVIDIA GEFORCE GTX 660

	HLL	Nvidia CSR	Nvidia HYB
raefsky2	16.31	9.37	4.42
af23560	15	7.2	14.7
mhd4800a	8.2	5.2	3.47
bcsstk17	11	8	7.2
lung2	7.62	3.59	6.8
pde100	13.7	6.13	14.3
FEM_3D_thermal2	14.56	7.5	15.77
mac_econ_fwd500	5.1	3.6	6.1

threads belonging to the same warp will read from the same hack using coalesced accesses and maximum performance, as long as the number of rows is sufficiently high and there is a low variance among sizes of compressed rows. The resulting pseudo-code to compute the matrix-vector product $y = Ax$ is shown in Fig. 6; in this case, AS and JA are accessed using *linear indexing*. The code is written taking into account a 0-based indexing and $threadIdx \in [0, blockSize)$ and $blockIdx \geq 0$. In Table I we present some performance results

```
function y=spMV(hackSize,hackOffsets,as,ja,nzr,x)
  idx = threadIdx + blockIdx*blockSize;
  hackId = idx / hackSize;
  hackLaneId = idx mod hackSize;
  hackOffset = hackOffsets(hackId);
  res = 0;
  for i=0:nzr(idx)-1
    ind = ja(hackOffset + i*hackSize + hackLaneId);
    val = as(hackOffset + i*hackSize + hackLaneId);
    res = res + val*x(ind)
  end
  y(idx) = res;
end
```

Fig. 6. Matrix-Vector product in HLL-G format.

of three different implementations of the spMV multiply routine: our HLL implementation, the CSR implementation from the official Nvidia cuSPARSE library [25], the HYB implementation also from cuSPARSE that aims to maximize performance on Nvidia hardware for general-purpose sparse matrices. The input matrices are taken from different real applications [26].

V. Enumerate Pattern

In Section IV we described the Vectorize pattern to define an index space over a set of data that is compliant, in terms of performance, with the GPU hardware. The goal of the Enumerate pattern is to define an index space on possible inputs, rather than to stored data, and let each thread generate and process them. This is usual in exhaustive search methods, also known as *brute-force* methods, in which we must enumerate all the candidate solutions for a particular problem and execute a test to verify if any of them satisfies a given condition.

Since the evaluation of each candidate solution is generally independent from all the others, an exhaustive search offers very good parallelism potential: the solution space can be partitioned in many ways with very few, if any, constraints. Indeed, the entire solution space can be either assigned to a single sequential process or partitioned and each part can be computed by a different process, possibly running on a different device.

Only a very small amount of data must be scattered at the beginning of the computation to each computing node, to allow the generation of the solutions' subspace; we must also collect periodically a fairly small amount of data from each device to eventually terminate the search if a stop condition is met (e.g., a satisfactory number of solutions has been found).

Although in many cases such subdivision seems trivial, some design considerations should be kept in mind when optimizing the performance on complex topologies of computing nodes with different computing power.

Problem

The problem is to design an exhaustive search algorithm in such a way that would ensure high efficiency and scalability over a set of many-core devices.

Solution

Given the set S of all possible solutions to a class of problems, we can execute an exhaustive search if there exist:

- a bijective function f from the set of natural numbers $N = \{0, 1, 2, 3, \dots\}$ into S, where S is either finite or countable;
- a test function $C : S \to 0, 1$.

Therefore, an exhaustive search involves the generation through $f(i)$ of all possible solutions, conducting tests with an increasing identifier i, and the test of the condition $C(f(i))$ for each entry. Note that $f(i)$ can be trivial or it can follow a heuristics to favor testing of the most likely solutions.

We define the operator *next* such that $\mathrm{next}(i, f(i)) = f(i+1)$. In many instances the execution of the *next* operator is much faster than the execution of $f(i+1)$, because it can be obtained with few manipulations of the $f(i)$ element's data.

Given the following cost functions:

- $K_f(i)$, the cost to generate a candidate solution from an identifier;
- $K_{\mathrm{next}}(i, f(i))$, the cost to generate a candidate solution from another candidate;
- $K_C(f(i))$, the cost to evaluate a candidate;

the cost K_{search} of an exhaustive search over a set of n possible solutions on a single process is:

$$K_{search} = K_f(i_0) + \sum_{i=i_0}^{i_{n-2}} K_{\mathrm{next}}(i, f(i)) + \sum_{i=i_0}^{i_{n-1}} K_C(f(i))$$

or if $\mathrm{next}(i, f(i)) \equiv f(i+1)$,

$$K_{search} = \sum_{i=i_0}^{i_{n-1}} (K_f(i) + K_C(f(i))).$$

If $K_{\mathrm{next}}(i, f(i)) < K_f(i+1)$ then the process' efficiency, defined as the time needed to test a solution over the time needed to generate the solution and then test it, will increase for larger n.

In a SIMT environment, a kernel can evaluate an entire interval of candidate solutions $[startID, startID + len)$ by passing the first input identifier $startID$ as argument, configuring the kernel's size to be equal to len, and generating i using $startID$ and the identifier of each thread and block. Then, the whole search space can be partitioned in multiple intervals to be computed in different kernel calls, using different $startID$ and optionally different grid size. A different partitioning of the search space in intervals can of course be provided, depending on the particular case.

The application of the Enumerate pattern is discussed in details in [7], where we presented an exhaustive key search on clusters of GPUs. In the same work we also proposed a hierarchical subdivision of the search space in order to distribute the search process over multiple nodes providing some metrics that can be used to obtain an optimal partitioning.

Consequences

In order to apply this pattern, both *next* and f functions should be implemented, with a subsequent increase in code complexity.

Case Study

We introduced the Enumerate pattern in [7], where we proposed a password cracking application as case study (nevertheless, the pattern can be applied to a wide set of exhaustive search algorithms). In this example, $f(i)$ generates all the strings of a given charset using the algorithm in Figure 7. Since the generation can be quite performance demanding, each thread evaluates a multitude of candidate solutions executing a single $f(i)$ and then applying the next operator defined in Figure 8, which in most of the cases just modifies the last character of the string in order to obtain subsequent solutions. The C function takes a string as input, evaluate its MD5 or its SHA1 hashcode, then compare the result with the hashcode

Require: $i \in \mathbb{N}, charset = ['a','b','c',\dots])$
Ensure: $str = f(i, charset)$
  $str = []$
  while $i > 0$ **do**
    $i \leftarrow i - 1$
    $currentCharId = i \bmod length(charset)$
    $currentChar = charset[currentCharId]$
    $str = currentChar \oplus str$
    $i = \lfloor \frac{i}{length(charset)} \rfloor$
  end while
  return str

Fig. 7. Pseudocode for the $f(i)$ operator; $\oplus$ is the string concatenation operator

Require: $str = f(id, charset)$
Ensure: $nextStr = f(id + 1, charset)$
  $nextStr = str$
  $currentPosition \leftarrow length(str) - 1$
  repeat
    $temp \leftarrow (id + 1) \bmod length(charset)$
    $nextStr[currentPosition] \leftarrow charset[temp]$
    $id \leftarrow \lfloor \frac{id}{length(charset)} \rfloor$
    $currentPosition \leftarrow currentPosition - 1$
    if $currentPosition = -1$ **then**
      $nextStr[length(str)] \leftarrow charset[0]$
      return $nextStr$
    end if
  until $temp = 0$
  return $nextStr$

Fig. 8. Pseudocode for the *next* operator

to crack. In Table II, we show how our password cracking application, which exploits the Enumerate pattern, achieves comparable or even better average performance results when compared to other popular software (BarsWF and Cryptohaze) over different architectures. The search space of this experiment was the set of strings made up to 8 alpha-numeric characters, including both lower and upper-case letters.

TABLE II
THROUGHPUT ON SINGLE GPU (MKEY/S)

	8600M	8800	540M	550ti	660
MD5 (theoretical)	83	568	359.4	962.7	1851
MD5 (our work)	71	480	214	654	1841
MD5 (BarsWF)	71	490	205	560	1340
MD5 (Cryptohaze)	49.4	316	146	410	1280
SHA1 (theoretical)	25	170	128	345	390
SHA1 (our work)	22	137	92	310	390
SHA1 (Cryptohaze)	20.8	132	68	185	377

VI. SORT AND PACK PATTERN

Problem

We need to build a structure including an arbitrary number of containers (buckets), each one filled by a variable number of elements. The destination bucket of each element is only known after a parallel computation on the elements. How to populate in parallel the buckets using a lock-free algorithm?

Forces

- the distribution of elements inside buckets is assumed to be random;
- the performance of the method should not depend on the number of elements per bucket;
- the memory footprint of the method should be sufficiently small to be contained in global memory;
- the insertion runtime should be as efficient as possible.

Solution

We know that each element can be inserted in only one bucket; therefore it is possible to pre-allocate two arrays:

- *elementIds*, storing the unique identifier of each element;
- *bucketIds*, storing for each element the unique identifier of the target bucket.

The main step (*sort*) of this pattern consists in sorting the *element/bucket* pairs, using the bucket identifiers as keys; at the end of this step, elements in the same bucket are stored sequentially in memory. The fastest sorting method on GPU reported in the literature is *radix sort*. It was first implemented by Satish et al. in [27], improved by Merill and Grimshaw in [28], and is included in the GPU Thrust library [29]; The radix sort implementation is based on the parallel prefix sum kernel, which constitutes itself the *Scan* pattern, belonging to the Transformation category in SIMPL.

The following step (*pack*) creates two additional vectors, with the same size of the set of destination buckets:

- *bucketStart*, in which the *i-th* element stores the index of the start of the *i-th* bucket in the *(elementIds, bucketIds)* array, or -1 if empty;
- *bucketEnd*, in which the *i-th* element stores the index of the end of the *i-th* bucket in the *(elementIds, bucketIds)* array, or -1 if empty;

Both arrays are initialized to -1; then we invoke a kernel which, for each thread, reads B_i and B_{i+1} from the *bucketIds* array. If $B_i \neq B_{i+1}$, then $bucketStart[B_{i+1}] \leftarrow i + 1$ and $bucketEnd[B_i] \leftarrow i$. The boundary conditions are $bucketStart[B_0] \leftarrow 0$ and $bucketEnd[B_{N-1}] \leftarrow N - 1$.

The contents of a bucket are retrieved by reading the entries in the arrays from the bucket start index to the bucket end index; the size of the bucket b is given $bucketEnd[b] - bucketStart[b]$. The solution is illustrated in Figure 9.

Consequences

The *sort* pass knows only the size of the input vector, and the method avoids the use of atomic functions; hence its performance is not affected by the distribution of the elements

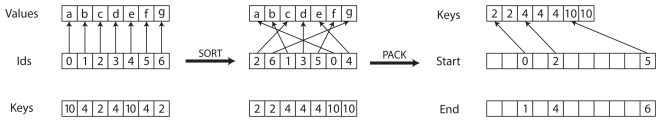

Fig. 9. Sort and Pack method for parallel insertion in a set of buckets

inside the buckets. Moreover, it scales with the number of elements rather than the number of buckets, since no empty container is considered in the sorting process. However, the size of *bucketStart* and *bucketEnd* is equal to the number of possible buckets; therefore the method is not applicable when the number of possible buckets is very large.

Example Uses

Among the many works that use this pattern to construct variable-sized lists we find [30], [31], [32], [33], [34].

VII. Conclusions

In this paper we presented a pattern language called SIMPL that can be used to accelerate and optimize a data-parallel algorithm on a SIMT architecture like a GPU. We started enumerating the difficulties and the objectives that arise when approaching parallel programming for the first time. Our pattern language was structured to answer those issues in the context of GPU programming. We classified five categories of patterns that aim answering to five different categories of difficulties. We then presented two example patterns from the Mapping category, providing for each one a case study that shows performance results in line with, or even superior, to those achieved by popular highly optimized software. We also described an example pattern from the Construction category, which is frequently applied in GPU kernels. As future work, besides publishing a full reference to the current language version, we will include those categories (Fault Tolerance, Energy Efficiency and Debugging/Profiling) that are missing at this time in SIMPL.

References

[1] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. of 1967 Spring Joint Computer Conf.*, ser. AFIPS '67 (Spring). ACM, 1967, pp. 483–485.

[2] J. L. Gustafson, "Reevaluating Amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[3] D. Barbieri, V. Cardellini, and S. Filippone, "Generalized GEMM kernels on GPGPUs: experiments and applications," in *Parallel Computing: from Multicores and GPU's to Petascale*, ser. Advances in Parallel Computing. IOS Press, Apr. 2010, pp. 307–314.

[4] ——, "Sparse computations on GPGPUs," DISP, Univ. Roma Tor Vergata, Tech. Rep. RR-12.90, Jan. 2012.

[5] ——, "Fast uniform grid construction on GPGPUs using atomic operations," in *Proc. of Int'l Conf. on Parallel Computing*, ser. ParCo 2013, Sep. 2013.

[6] D. Barbieri, V. Cardellini, S. Filippone, and D. Rouson, "Design patterns for scientific computations on sparse matrices," in *Euro-Par 2011: Parallel Processing Workshops*, ser. LNCS. Springer, 2012, vol. 7155, pp. 367–376.

[7] D. Barbieri, V. Cardellini, and S. Filippone, "Exhaustive key search on clusters of gpus," in *Proc. of 28th IEEE Int'l Parallel & Distributed Processing Symp. Workshops*, 2014, pp. 1160–1168.

[8] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*, ser. Center for Environmental Structure Series. Oxford University Press, 1977.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] D. Garlan and M. Shaw, "An introduction to software architecture," Pittsburgh, PA, USA, Tech. Rep., 1994.

[11] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, 2nd ed. New York, NY, USA: John Wiley & Sons, Inc., 2000.

[12] J. Anvik, J. Schaeffer, D. Szafron, and K. Tan, "Why not use a pattern-based parallel programming system?" in *Proc. of Euro-Par 2003*, ser. LNCS, vol. 2790. Springer, 2003, pp. 81–86.

[13] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*, 1st ed. Addison-Wesley Professional, 2004.

[14] B. Wilkinson, J. Villalobos, and C. Ferner, "Pattern programming approach for teaching parallel and distributed computing," in *Proc. of 44th ACM Tech. Symp. on Computer Science Education*, ser. SIGCSE '13. ACM, 2013, pp. 409–414.

[15] M. D. McCool, "Structured parallel programming with deterministic patterns," in *Proc. of 2nd USENIX Conf. on Hot Topics in Parallelism*, ser. HotPar'10, 2010.

[16] J. L. Ortega-Arjona, *Architectural Patterns for Parallel Programming: Models for Performance Estimation*. Saarbrücken, Germany: VDM Verlag, 2009.

[17] B. L. Massingill, T. G. Mattson, and B. A. Sanders, "SIMD: an additional pattern for PLPP (Pattern Language for Parallel Programming)," in *Proc. of 14th Conf. on Pattern Languages of Programs*, ser. PLOP '07. ACM, 2007.

[18] H. Gonzlez-Vlez and M. Leyton, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," *Software: Practice and Experience*, vol. 40, no. 12, pp. 1135–1160, 2010.

[19] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991.

[20] S. Siu, M. D. Simone, D. Goswami, and A. Singh, "Design patterns for parallel programming," in *Proc. of 1996 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, 1996, pp. 230–240.

[21] R. P. Gabriel, *Patterns of Software. Tales from the Software Community*. Oxford University Press, 1996.

[22] S. Filippone and M. Colajanni, "PSBLAS: a library for parallel linear algebra computations on sparse matrices," *ACM Trans. Math. Softw.*, vol. 26, no. 4, pp. 527–550, Dec. 2000.

[23] S. Filippone and A. Buttari, "Object-oriented techniques for sparse matrix computations in Fortran 2003," *ACM Trans. Math. Softw.*, vol. 38, no. 4, 2012.

[24] D. Barbieri, "spGPU: Sparse Matrices on GPU," https://code.google.com/p/spgpu/.

[25] NVIDIA Corporation, "CUDA cuSPARSE library," 2015, http://developer.nvidia.com/cusparse.

[26] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Nov. 2011.

[27] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Proc. of 2009 IEEE Int'l Symp. on Parallel & Distributed Processing*, ser. IPDPS '09, May 2009.

[28] D. G. Merrill and A. S. Grimshaw, "Revisiting sorting for GPGPU stream architectures," in *Proc. of 19th Int'l Conf. on Parallel Architectures and Compilation Techniques*, ser. PACT '10, 2010, pp. 545–546.

[29] N. Bell and J. Hoberock, "Thrust: a productivity-oriented library for CUDA," in *GPU Computing Gems, Jade Edition*, W. W. Hwu, Ed. Morgan Kaufmann, Oct. 2011, ch. 16.

[30] S. Green, "CUDA particles," NVIDIA Corporation, Tech. Rep., 2008.

[31] L. Luo, M. D. F. Wong, and L. Leong, "Parallel implementation of R-trees on the GPU," in *Proc. of ASP-DAC '12*. IEEE, 2012, pp. 353–358.

[32] J. Kalojanov and P. Slusallek, "A parallel algorithm for construction of uniform grids," in *Proc. of 1st ACM Conf. on High Performance Graphics*, ser. HPG '09, 2009, pp. 23–28.

[33] S. Le Grand, "Broad-Phase Collision Detection with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley Professional, Aug. 2007, ch. 32.

[34] J. Bédorf, E. Gaburov, and S. Portegies Zwart, "A sparse octree gravitational n-body code that runs entirely on the GPU processor," *J. Comput. Phys.*, vol. 231, no. 7, pp. 2825–2839, Apr. 2012.