Shannon meets Cray

From circuits complexity to SIMD-programs

Partner	Name	Position	Role	Involvment
CRIStAL (U. Lille)	Charles Paperman	MCF	**PI**, **Lead** (WP2, WP5, WP6)	30
CRIStAL (U. Lille)	Sylvain Salvati	PU	WP2, WP3, WP5	6
CRIStAL (U. Lille)	Michael Hauspie	MCF	WP1, WP2, WP4, WP6	6
CRIStAL (U. Lille)	Antoine Limasset	CR (CNRS)	WP2, **Lead** WP7	6
LIP (ENS Lyon)	Gabriel Radanne	CR (INRIA)	**Lead** (WP1, WP3, WP6)	24
LIP (ENS Lyon)	Matthieu Moy	MCF	WP3, WP4	6
LIP (ENS Lyon)	Ludovic Henrio	CR (CNRS)	WP3	6
LIP (ENS Lyon)	Denis Kuperberg	CR (CNRS)	WP3, WP5	6
LCIS (Grenoble INP)	Laure Gonnord	PU	WP1, WP3, **Lead** WP4,	12
LCIS (Grenoble INP)	Valentin Egloff	MCF	WP1, WP4	6
LCIS (Grenoble INP)	Bruno Ferres	MCF	WP1, WP4	6
LaBRI (U. Bordeaux)	Nathanaël Fijalkow	CR (CNRS)	WP1, WP4, WP5	12

I Proposal's context, positioning and objective(s)

Stream processing is at the heart of many applications. For instance, it is used by internet providers who scan network traffic to detect suspicious behaviors. More generally, crawling huge textual data is pervasive and concerns both raw and structured texts like XML or JSON documents. Bioinformatics is another domain that demands efficient stream processing, for example for finding patterns in genomes, or comparing DNA sequences. In each of these problems, the basic buildings blocks are simple, but combining them to efficiently treat terabytes of data is far from straightforward.

Facing data that gets larger and larger, hardware manufacturers keep on increasing CPUs computing power. Apart from increasing clock rates, manufacturers try to leverage *bit-level parallelism*; e.g. *regularity* in data offers opportunities of applying the *same operations* to *multiple inputs simultaneously*. These particular parallel programs are often coined as *vectorial programs*, a notion proposed in the seventies by Leslie Lamport [23] and notably implemented in the Cray series of super-computers [52]. Modern mainstream architectures (e.g. `X86_64`) also support this style of programming via *Single Instruction Multiple Data* (SIMD). SIMD instructions act on registers where several machine words are packed, and apply an operation on each of the words in the register.

SIMD instructions met success in linear algebra and scientific computing. In this setting, compilers can now automatically transform programs operating on "*regular data*" into vectorial ones using *auto-vectorization* [14, 32, 34, 48], improving performance by several orders of magnitude. Auto-vectorization is mostly based on finding loops in programs that can be executed in parallel, and then generate vector instructions that can be executed on multiple data elements at once.

Usual HPC kernel operates on multidimensional data with limited branching, which is well exploited by auto-vectorization compilation techniques, dedicated libraries and scientific computing libraries. By opposition, many streaming applications mentioned above operate on sparse one-dimensional data with highly branching treatment. Those application can be heavily accelerated through data-parallelism but auto-vectorizers fail to take advantage of SIMD-instructions, leading expert programmers to craft efficient implementations by hand. Furthermore, the data-parallelism considered is very thin grained, which is poorly handled by tools such as OpenMP. For instance,

```c
char * custom_memchr(char *s, char c, size_t n){
   for (size_t i=0; i<n; i++)
      if (s[i] == c) return (char *)(s + i);
   return NULL;
}
```

Figure 1: Vanilla `memchr` implementation

```c
char * custom_memchr(char *s, char c, size_t n){
   vector mask_c = load_mask(c);
   for (size_t i = 0 ; i < n ; i += vector_size) {
      vector chunk = uload(s + i); // load the chunk of the stream
      vector c_in_chunk = vectorEq(chunk, mask_c); // compare bytewise
      mask offsets = maskFromVector(c_in_chunk); // compute the bitmask
      // ctz returns the offset of the left most 1 in mask
      if (offsets != 0) return (char *)(s + i + ctz(offsets));
   }
   // a sequential variant for the end of the bloc.
   for (size_t i= n - (n % vector_size); i<n; i++)
      if (s[i] == c) return (char *)(s + i);
   return NULL;
}
```

Figure 2: Vectorial implementation of memchr

the popular SimdJSON library [24] and the PI's project Rsonpath [11] use handmade clever tricks to achieve several orders of magnitudes speedup compared to classical implementations. `Rsonpath` achieves up to 4GB/s throughput while non-SIMD variants top out at 0.2GB/s, even though both admit an asymptotic linear-time complexity in the stream size.

Similarly, in Bioinformatics, one of the heaviest computational operation is to align the output of a sequencing machine to DNA sequences [43]. Again, state of the art tools rely on hand-crafted dedicated SIMD implementations [25]. Such implementation are all the more difficult to produce as they require writing assembly-level code tailored for each target architecture. Even for a given architecture, crafting the exact series of instructions that give best performances is highly non trivial. Understanding, reading and maintaining such programs is hard.

An introduction to SIMD programming To demonstrate the typical optimization provided by SIMD (Single Instruction, Multiple Data) on modern hardware and the sheer difficulty of writing code using these instructions, let's consider the `memchr` function as an example. This function searches for the next occurrence of a character in a memory buffer and returns a pointer to it. If the character does not exist in the buffer, it returns a null pointer. This function is commonly used to search elements in text and is present in C standard libraries since the early 80s.[1]

The vanilla implementation, displayed in Fig. 1, already reaches performance of several GB/s with traditional compiler optimisations in mainstream compilers, such as GCC. It can nevertheless be largely improved using dedicated SIMD instructions. On common CPUs, these instructions are available through *intrinsics*, C-level functions that are recognized by compilers to emit a given CPU-specific instruction. These instructions have a complex nomenclature which makes code hard to read.

Taking advantage of *intrinsics*, the code in Fig. 2 is a simplified version of the actual implementation `memchr` in `glibc` (which is too long to fit in this document, several hundreds of assembly code lines). This simplified code is already much more involved than the vanilla version of `memchr`. We now give a bird's eye view of the algorithm. First, we create a *mask*, on line 2, and store it into a SIMD register, denoted by the type `vector`. Then we look at the input chunk by chunk in order to

[1] memchr in Unix V8 ('84) https://www.tuhs.org/Archive/Distributions/Research/Dan_Cross_v8/v8.tar.bz2

search for the character `c` in a whole chunk, using the loop line 3. For this purpose, we first use SIMD comparisons, then store the position of the first 1 bit, of which gives the position of the possible first occurrence of `c`. We finalize the code by handling the end of stream sequentially, on line 13. The complexity of this code is worth the effort. Benchmarks, summarized Table 1, show that the throughput is about 7 times better with the vectorized version of Fig. 2 than with the vanilla version of Fig. 1. Also observe that the extra complexity `glibc` gives it a further edge: with functions as widely used as `memchr`, every gain in efficiency has an impact in the global efficiency of many programs.

Implementation	Code Fig. 1	Code Fig. 2	`glibc`
gcc (GByte/s)	2.9	20.2	21

Table 1: Throughput comparison of `memchr` implementations on a personal computer.

a Objectives and research hypothesis

Writing programs that benefit from vectorization is demanding in time, high-level expertise and thus comes with great costs. Moreover, the know-hows are scattered among numerous fields and are often fleshed into pieces of codes that pertain to system architectures, compilers or certain specialized industrial programs. In a nutshell, vectorization is far from being available for day-to-day programming. The end goal of the project is to ease the use of SIMD instructions in the context of stream processing. The outcome will be of two forms: concepts to harness bit-level parallelism in a systematic way, and practical high-level tools to program with SIMD instructions.

A key to achieve desired efficiency for data-processing programs is to carefully craft the control flow and to streamline it into variations of finite state machines. To optimize it further, the finite state machine itself must used bitlevel paralelism. As it turns out by preliminary research, the potential for bitlevel parallelism for a finite automaton can be model by its *circuit complexity* [30]. Circuit complexity, introduced by Shannon [42] in the early days of computer science, offers a powerful framework to study parallelism. Notably, Straubing [45] built a powerfull theoretical framework, based on finite model theory and semigroup theory, to study the parallel complexity of regular languages.

We have conducted preliminary research that already tightens the relationship between SIMD-programs and circuit complexity. The notion of *vectorial circuits* [41, 36] is the missing piece that articulates Shannon's notion of circuits and SIMD instructions or their former incarnation as Cray series of super-computers. This notion is where *Shannon meets Cray*: vectorial circuits formalize a specialized model of computation which operates on *bit-vectors* of arbitrary sizes and closely match the ability of SIMD CPUs. They have several advantages: they convey an inner notion of parallelism close to bit-level parallelism, connect circuit complexity to vectorization, and seem to support a compilation process akin to synchronous programming.

Exploring, extending and exploiting this theoretical interplay between sequentiality, bit-level parallelism and circuit complexity is the main drive of the project. For this work to bear fruits, it will be blended with the methodologies of programming language design and compilation techniques for parallel architectures. The overall validation of the project will take the form of applications to data processing and Bioinformatics. To sum up, our project aims to:

- Section a.1: Develop the foundation of vectorial circuits and connect them with stream processing,
- Section a.2: Design a domain-specific programming language for stream processing that:
  - provides powerful abstractions to write concise programs,
  - has a low level interpretation in vectorial circuits,
  - integrates in existing languages used by high-performance programmers.
- Section a.3: Confront results of streaming processing with applications in {Data, DNA}-processing.

a.1 A model for streaming vectorial programs

We will model vectorial programs in the streaming context through the introduction of VIR: a Vectorial Intermediate Representation for streaming algorithms. VIR will encompass many operations useful for designing vectorial programs (branches, byte manipulation, stream transformation). Vectorial circuits are VIR's restriction to bitmask manipulation. Vectorial circuits will be the place where code simplification and optimization is done.

> *Example 1.* To illustrate vectorial circuits, consider the task of finding XML opening tags in a stream, i.e. finding substrings matching the regular expression $\mathcal{T} =$ `<[a-Z]*>`. As it turns out, there is a highly efficient vectorial implementation of this program. Given an input word, let $\vec{o}$, $\vec{c}$ and $\vec{a}$ the bitmasks of the positions in the input for all '`<`'s, '`>`'s and alphabetical letters respectively. The vectorial circuit $((\vec{o} + (\vec{o}|\vec{a})) \wedge \vec{c})$ produces a bitmask of the last position of each opening tag. This program only uses bitwise instructions ($\vee$, $\wedge$ and $\neq$) and carry-propagated addition on streams. It is possible to convert this circuit into code by evaluating the addition chuck by chunk simultaneously with the other operations while propagating the carry.

In this example, the equivalence between the sequential program (i.e. the regex) and the vectorial circuit is far from obvious. We have recently showed [36] how certain classes of regular expressions can be systematically compiled to vectorial programs based on similar tricks. Going beyond these particular classes of regular expressions and using a richer set of circuit operations is a significant *theoretical* challenge undertaken by **Shannon meets Cray**.

Expressivity of vectorial circuits The expressivity of vectorial circuits depends mostly on the set of basic operations allowed on bitmasks. Modern CPUs offer complex bitmasks operations: prefix-XOR, counting bits in bitmask or complex lookup table.

> *Example 2.* prefix-XOR outputs at each position the XOR of the bits in the prefix up to that position. This can notably be used to accelerate identification of words within delimiters (by prefix-XOR of the bitmask identifying the delimiters). From a theoretical standpoint this operation is costly: it requires circuits falling outside the complexity class AC^0 [10], meaning it cannot be implemented with only bitwise Boolean operations and addition. It can however be implemented with Carry-Less multiplication instructions [51], offering a powerful generalisation. Vectorial circuits using this operation have never been studied and offer interesting new venues to vectorize streaming programs.

Fixing a set of operations, **Shannon meets Cray** aims to capture the expressivity of vectorial circuits:

- **Sequentiality**. Building finite-state machines that capture the expressivity of the vectorial circuits that use the fixed operations.
- **Parallelism**. Analysing in which circuit complexity classes these vectorial circuits are.

This line of work will lay the foundations to compilation procedures to vectorial programs in several ways. Firstly, it will allow targeting classes of queries in some languages (e.g. regex, or XPath or JSONPath) which can be accelerated with vectorization. Then we will be in position to design dedicated compilation procedures. Secondly, it will set the theoretical stage to abstract away hardware diversity. Thirdly, it will clarify which fragments of VIR can be compiled with which instructions sets.

Compilation of VIR VIR aims to be a portable abstraction of SIMD programs. A main milestone for the application of VIR will be the construction of a complete compilation pipeline from VIR to several standard SIMD-architectures (`X86_64`, `AVX-2`, `AVX-512` and `ARM Neon`). We shall also include some more experimental instructions sets such as the `RISCV "V"` extensions [1]. Our focus on stream processing allow us to exploit the similarity between VIR and synchronous languages such as Lustre, notably the transformations from *data flow* to *control flow* which form the backbone compilation of synchronous languages. One of the main challenges will be to mix the vectorial features with this main source of inspiration, and especially express the hybridisation between sequential and vectorial compilation. We plan to work in progressive steps: First formally define and implement the core

compilation process. Then progressively extend the VIR, both its synchronous and vectorial parts. Finally enhance the efficiency of the generated code.

a.2 Vectoid: a language for streaming algorithms

While they make for a powerful intermediate representation, vectorial circuits are difficult to program with: they are too low-level and rely on a paradigm unfamiliar to most programmers. The second aim of this project is to allow more programmers to harness the efficiency offered by SIMD instructions. For this purpose, we will design and develop a domain specific language, dubbed Vectoid, which compiles to SIMD-programs via the Vectorial Intermediate Representation.

The Vectoid language Supported by the work on the VIR, we will design Vectoid with *real life use cases* in mind. Specifically, Vectoid should allow existing streaming text algorithms to be expressed and executed in the runtime model presented above. For inspiration, text-based analysis tools for JSON, XML, CSV, or even *coreutils* (`cut`, `wc`, `tr`, ...) and regex tools, have hand-crafted optimized SIMD versions. Similarly, Bioinformatics computational tasks require heavily optimized streaming procedures around the numerous variations of the alignment problem [43, 25, 8]. We will produce a systematic survey of SIMD algorithmic techniques found in practical settings in different communities in order to discover what programmers would need to write efficient streaming algorithms. The accumulated knowledge will drive the design and compilation of Vectoid.

Example 3. We consider again the implementation of `memchr` mentionned in Section I, using a prospective syntax for Vectoid. Notice that the program is particularly concise and only retain high level operations. Let's have a closer look. On line 1, we specify the arguments: `c` a single `Byte`, and `B` a stream of bytes. On line 2, we use stream-comprehension similar to list-comprehension in other languages, such as Python. This comprehension shall be compiled to a streaming SIMD-implementation using the `uload` and the `maskFromVector` function from Fig. 2. The `Index` operator on line 3 returns the offset of the first 1 in the bit-string, compiled to an efficient sequential search.

```
1
2
3 def memchr(B: [Byte], c: Byte) -> Integer:
4     let a: [Bit] = [(e==c) for e in B]
5     return Index(a)
```

Example 4. Let us now reconsider the task of finding opening tags in an XML document. The input B is a list of bytes (`[Byte]`). As the original description in Section a.1, we define three bitmasks and combine them. Here, we borrow operators from temporal logic (`Next` and `Until`) which can be compiled to efficient SIMD code. Note the importance of keeping streams synchronised.

```
1 def match_tag(B : [Byte])
2     let open_quote: [Bit] = [ e == '<' for e in B ]
3     let close_quote: [Bit] = [ e == '>' for e in B ]
4     let alpha: [Bit] = [ e in [:alpha:] for e in B ]
5     return Next(open_quote, Until(alpha, close_quote))
```

As shown, Vectoid describes *kernels* that capture the core of the computation. We aim to *embed* such kernels into existing host languages, as it greatly improves usability and diffusion of the language, allowing many programmers to use Vectoid to define SIMD-powered code and distribute it into existing ecosystems. We shall thus pay attention to the integration of Vectoid in host languages such as `C` and `Rust`. This is highly non-trivial, as it requires interfacing between SIMD and non-SIMD code.

Vectoid will be a *functional*, *modular*, *statically typed* programming language with a *limited expressivity* that fits the model of vectorial circuit mentioned previously. Functional programming, which manipulates structured and *immutable* data, lends itself particularly well to highly optimizing compilation of domain-specific languages [18, 46, 12]. Static typing, in turn, ensures precise compile-time

analysis, allowing for more aggressive compilation. *Modularity*, allows programmers to adapt modern programming practices and develop Vectoid libraries. We also expect this last task to be challenging.

Naturally, this language will come with a formal description that details its semantics and its link with VIR. It will be developed hand-in-hand with VIR, both for its formal study and its compilation.

Efficient compilation of Vectoid

The ultimate goal of Vectoid is to produce highly efficient programs. As Vectoid programs will be compiled to Vectorial Intermediate Representations, the compilation of Vectoid and VIR will have strong interactions.

Example 5. Consider the program which compares two streams, illustrated below. This program requires synchronisation between streams: A and B are not known to be synchronized and the compilation procedure for the synchronized walk in line 2 requires a particular care. Line 3 introduces further desynchronization with the `when` keyword, which emits an element only when b is true.

```
1 def diff_offsets(A: [Byte], B: [Byte]) -> [Integer]
2   let u: [Bit] = [ (a != b) for a in A and b in B]
3   return [ pos for pos, b in enumerate(u) when b ]
```

Figure 3: Vectoid program to output the positions of all differences between two streams

Synchronous languages [16], have the ability to express these kind of synchronization on events via *clock* calculus. Such languages also allow efficient compilation and generation of communication buffers. We will thus take major inspiration from existing synchonous works.

a.3 Application to streaming

Shannon meets Cray envisions the developments of theory, technology and applications as an integrated effort. Vectorization of general programs is a long-standing open problem. Our strategy to attack this problem consists in choosing a restricted class of programs that has a large number of applications: stream processing. Among many possible applications in streaming, we target Data and DNA processing. Solving difficult applied problems in these areas will allow us to validate our developments on VIR and Vectoid. The project will incorporate from its very beginning demands and constraints coming from these fields. Coping with exascale problems is a common challenge they face. We expect several outcomes from these interactions: guidance from early experiments to theoretical and technical developments, validation of our approaches, flexible research strategy and high impact in these important application areas.

For the moment, Data and DNA do not benefit from automatic SIMD-optimizations. Such development are costly, both in term of knowledge and time, and must be carried out anew for each new hardware and new algorithm. By improving portability to new hardware and ease of writing, Vectoid will tremendously reduce these costs and thus pave the way to new applications.

In the case of Data processing, we observe that most research is conducted by applying tailor made SIMD-optimization techniques. There does not seem to be any systematic methodologies and these results mostly constitute engineering *tour de force*. In the more specific context of query languages, software is mostly designed by people with no knowledge in SIMD-optimization. This leads to different problems. First optimizations are strongly specific to their application and therefore cannot be applied to other problems. Second the code suffers from the aforementioned problems of maintenance and portability. By isolating fragments of query languages with efficient SIMD-optimization, **Shannon meets Cray** will allow the implementation of query execution engines that are efficient *by design*.

Bioinformatics is facing a data deluge and rapidly changing DNA sequencing technologies. These evolution put a lot of burden on programs sequence comparison tasks. There is a clear need to enable rapid implementation of innumerable variations of DNA-alignment algorithms that are both efficient and portable. Vectoid will meet this demand. We hope to reduce the development costs and thus unlock developers' creativity as they will be relieved from the technical burden of SIMD-optimizations.

b Position of the project as it relates to the state of the art

Automatic vectorization CRAY-1 supercomputer was the first machine reaching the hundreds of MFLOPS thanks to its vectorial design. It was shipped together with a Fortran compiler equipped with auto-vectorization [40]. Already back then, loops with control flow where identified as a challenge for further works. This gave rise to the motion of data-dependencies [2] and numerous code-rewriting techniques, including the polyhedral model [9, 48]. Nowadays, all modern compilers (GCC, LLVM and ICC) are equipped with auto-vectorizers which are under continuous development and improvement [33, 19, 13]. Intel provides guidelines [21], which we believe illustrate well the state of the art of auto-vectorization altogether:

- Use simple for loops. Avoid complex loop termination conditions – the upper iteration limit must be invariant within the loop. For the innermost loop in a nest of loops, you could set the upper limit iteration to be a function of the outer loop indices.
- Write straight-line code. Avoid branches such as switch, goto, or return statements; most function calls; or if constructs that cannot be treated as masked assignments.
- Avoid dependencies between loop iterations or at the least, avoid read-after-write dependencies.

Some libraries try to make help the programmer to code SIMD-programs via thin abstraction over the assembly (e.g. the IPCS compiler or the rust vectorial stdlib). Such libraries do not provide high-level coding abstraction, which is the goal of Vectoid. Other provides pre-packaged algorithms implemented with SIMD in application domain (e.g. ML acceleration like OpenCV). These are a great source of inspiration for Vectoid, and potential end users. Finally, it is possible to produce efficient SIMD through alternative compilation methodology. For instance, OpenMP targets coarse-grained parallel tasks, and can locally accelerate simple loops with SIMD. Alternatively, the polyhedral methodology [17] has been generally used to optimise dense regular loop nests, either in C or with a DSL. As far as we know, none of the existing compilation-based techniques efficiently handle the streaming application we're focusing on. In our scenario, we traverse data with a complex control flow, which often lies beyond the scope of these methodologies.

Languages for parallel and streaming computation Restricting the syntax of a programming language for a domain specific purpose can open the room to many optimisation. Hence, several programming languages have emerged to target specific optimisation into parallel architectures from academic works (e.g. [46, 6, 39]) or from industrial domain (e.g. OpenCL). Similarly, many languages capture streaming processing specificity to offer highly efficient compilation [12, 47]. So far, such languages either target array-like tasks and take advantage of SIMD, or target streaming applications, but do not combine both. Synchronous [16] and/or dataflow [4] languages also deal with streaming computations, for different purposes such as reactive programming or signal processing. Their long history has lead to many improvement of the initial code generation, but to our knowledge there is no SIMD backend at the time of writing.

SIMD and data, DNA-processing SIMD has been extensively used to improve data-processing tasks such as text searching and DNA-processing. Myers [31] introduced bit-vector manipulation for approximate string matching which had a great impact on stringology and DNA-processing. Hyperscan [50] project by Intel was a successful effort to provide a vectorial implementation of a regex engine, and its lessons have been included in Rust regex engine. Several projects have been developed to improve parsing [27, 24] and more recently querying JSON [22, 11]. For relational databases, SIMD-optimizations of query engine are being studied [15].

Model of SIMD optimizations From a theoretical point of view, SIMD-programming has been modelled by vector machine [38], by parallel Vector model (a variation around the RAM model with vector abilities [5]), and more recently by vectorial circuits [41, 36], which actually can be seen as a simplification of vector machines.

Preliminary results The PI with Michaël Hauspie and Sylvain Salvati have studied SIMD optimization of regular expressions in the context of the PhD of Claire Soyez-Martin. This leads to a first article on algebraic methods to compile regular expression into vectorial code [36] and an ongoing implementation in Rust. This implementation has provided the consortium with preliminary benchmarks about the potential optimizations that those methodology could bring in the scope of regular expression optimization. The PI has co-advised with Filip Murlak, the master thesis of Mateusz Gienieczko from Warsaw university on SIMD implementation of a fragment of the JSONPath query language [3], which led to the *rsonpath* project [11], currently the fastest query engine for JSONPath.

Novelty and originality Although many domain-specific languages are available to simplify the design of parallel programs, few of them address SIMD optimizations for streaming algorithms. Since CRAY-1, significant research on optimizations focused on regular nested loop kernels operating on arrays, following the need of scientific computing. While significant progress has been made since the 1980s, notably using the polyhedral model [9], optimization for branches within a single loop has not been investigated as thoroughly, as illustrated on `memchr` in Section a.2. These optimizations cannot handle tasks that are intrinsically sequential, despite the existence of efficient code for them. In our previous example (see Example 1), we illustrate how intricate use of bitmask operations can encode a portion of a program's sequentiality. This kind of optimizations falls within the realm of logic and finite automata, which, to the best of our knowledge, has never been used in the context of SIMD optimization.

As part of this project, we have identified two challenging fields of application: data processing and Bioinformatics. The specific outcome we foresee is the production of code with less engineering effort, easier maintainability and better portability to different architectures. By the end of the project, when the work on Vectoid bears its fruits in these areas, other applications of the language will be easily reachable. More importantly, the results of **Shannon meets Cray** will allow researchers to focus on the algorithmic content of the vectorization rather than on technicalities. Furthermore, certain optimizations are difficult to uncover, even for seasoned engineers. The ground work that underpins Vectoid will propose and spread presently unknown optimizations and thus break some computational barriers of these fields.

c Methodology and risk management

The **Shannon meets Cray** project is divided into several work packages split over a 5 years period: a one year preparatory phase, a three years main research phase and a one year project completion phase.

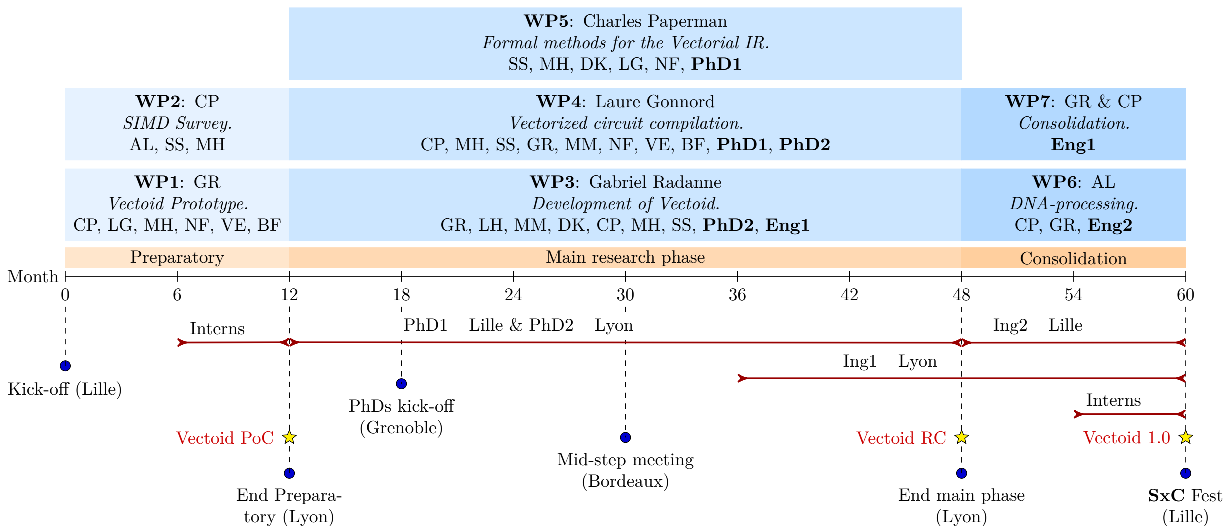

Figure 4: Gantt of the project

The preliminary phase of the project involves Work Packages (WP) 1 and 2, which aim respectively to establish the foundation of the project by providing a minimal implementation of the language (Vectoid PoC) and a normalized set of examples. The main research phase is divided into three WPs, focusing on the design, compilation, and expressiveness of the programming language, as well as on improving the compilation process using a verification-based methodology. The main outcome of the main research phase will be the Vectoid release candidate (Vectoid RC). Finally, in the consolidation phase, WP 6 aims to mature the language and technological development while testing its performance for data-processing applications, and WP 7 aims to develop applications in Bioinformatics, specifically in DNA processing. This last phase of the project represents the culmination of our research efforts, with the release of (Vectoid 1.0). Several consortium meetings will be organized during the project. The organisation is summarized in Fig. 4.

Software development Our project's main deliverable is a language and compiler, which require dedicated software development. We plan to develop in the Rust ecosystem due to its expressive pre-compilation macro system which fits our project needs, and its rich and open ecosystem. Additionally, we have experience with pre-project development using Rust, specifically with the *rsonpath* and *regex macros projects.* Rust's tooling enables us to make libraries developed in Rust available to other language ecosystems easily. Vectoid will be released under a permissive open-source license, both for the wider research community to use and improve, and to promote its large scale adoption.

Main research phase organization The main research phase of our project consists of three complementary work packages: WP 3, WP 4, and WP 5, which form the core of our work. WP 3 primarily focuses on language design and high-level compilation phases to VIR. WP 4 aims to define and compile the VIR high-level representation to various backends, handling the transformation to produce the flow semantics of the code. These two work packages are in tandem, collaborating closely on the specification of the VIR intermediate representation. Finally, WP 5 focuses on the formal expressivity of VIR to provide a foundational study and understand its inner expressivity. This work package will also drive applications in data-processing.

Data handling and benchmarking methodology Throughout the project, we want to test our development of both VIR and Vectoid against concrete benchmarks and applications. These benchmarks are crucial to drive the project in the appropriate direction and will be highly sensitive to the hardware used. To ensure accuracy and reproducibility, we plan to rely on Grid5000, which provides access to a diverse range of CPU architectures and is available to members of the consortium. Additionally, we will make the dataset used for the benchmarking, the code, and details about the hardware publicly available through an accessible repository and on the project webpage to ensure transparency and enable other researchers to verify and reproduce our results. Finally, some objectives of Vectoid target data-processing tasks. Those tasks could benefits from SIMD-optimisation even when depending on I/O intensive context (e.g. reading from disk). Indeed, some modern hard-drives reach bandwidth of several gigabyte per seconds which will not be topped by vanilla sequential implementation. The LINKS team (INRIA Lille) possesses IO-oriented benchmarking server with fast NVMe disks which will be used to showcase the effectiveness of Vectoid in disk-intensive operations.

WP 1. Vectoid prototype.

Participants: G. Radanne (Lyon); C. Paperman, M. Hauspie (Lille); L. Gonnord, B. Ferres, V. Egloff (Grenoble) ; N. Fijalkow (Bordeaux)

This first WP aims at bootstrapping the project by creating a first minimal implementation of Vectoid. This prototype will contain a first sketch of the front-end, allowing us to write stream processing programs, and a simple version of our compilation pipeline, allowing us to experiment with optimisations and instruction sets. The goal of this early prototype is to make a rough sketch of all

the pieces that will be refined in the future Work Packages, to lay down expectations for the project, and to showcase early results for publication.

To limit the amount of work, we will provide minimal integration with the host programming language, which will only be Rust initially. Compilation target will be restricted to X86_64 with the AVX2 instruction set. Conditionals and functions will be limited.

All team leaders will contribute to this WP to ensure the appropriate bootstrap of the project. The expertise of G. Radanne and L. Gonnord will be relied upon for language design and compilation. C. Paperman will be responsible for integrating Vectorial circuits optimization, while M. Hauspie will handle the integration into the host programming language. N. Fijalkow will be responsible for formal semantics and the relationship with logical fragments and automata theory. Interns will help at the evaluation of the prototype at the second half of the WP.

Task 1.1. First version of Vectoid. As presented in Section a.2, our programming language, dubbed Vectoid, is strongly typed, with functional operators that map to SIMD operations. The minimal viable Vectoid prototype will offer a rather simple type system (for instance, simple constant types such as Bytes and Integer, and sequences of those primitive type). It will provide the basic streaming operators that we have seen in the previous examples, but without handling of complex desynchronized streams, as shown in Figure 3. The output of a Vectoid programs will be very constrained to simplify its interaction with the host programming language.

Task 1.2. Early compilation pipeline. This early prototype is a mean to experiment with the compilation pipeline. Initially, we will provide standalone SIMD-implementations for Vectoid operators and compose them, with limited optimisations, in the style of copy-and-patch techniques [54]. The amount of specific optimizations performed on the code will be kept minimal. The focus will not be on raw performances but rather on conciseness of the compiler.

Task 1.3. White paper. During the development of the Vectoid prototype, we aim to not only gain knowledge about SIMD-programming but also to refine the scope of the **Shannon meets Cray** project's final language. We will consolidate this knowledge and the projections for the initial release of Vectoid into a comprehensive white paper, which will be available on a public repository and potentially submitted for publication.

WP 2. {DNA, DATA}-processing survey.

Participants: C. Paperman, S. Salvati, M. Hauspie, A. Limasset (Lille)

SIMD optimisations are used in a large variety of settings, scattered in various fields and technologies. We need to put this knowledge in shape to be usable to improve the design of Vectoid. This WP relies on the Lille center and its diverse set of expertise in data-processing, security and systems and Bioinformatics. Interns will help at building dataset of the second half of the WP.

Task 2.1. A review of academic literature. Our review will focus on papers that utilize SIMD to enhance the performance of algorithm implementations, with a specific emphasis on data and DNA processing tasks. There is an overwhelming amount of papers and optimized SIMD code available, and we acknowledge that a systematic review is not feasible. We will focus on the fields of programming language and compilation (OOPSLA, PLDI, CGO, ASPLOS, …), data-processing (VLDB, ICDT, SIGMOD) and DNA-processing (RECOMB/ISMB, SPIRE, Nature Methods) but also Security conferences (USENIX Security, IEEE Security & Privacy) as efficient stream processing is an essential aspect of intrusion detection. Finally, operating system conferences (OSDI, SOPS, Eurosys, …) bring us insights on hardware related issues we could encounter while leveraging SIMD instructions set.

Our objective is to perform a thorough and systematic analysis of the papers we select, with the aim of providing valuable insights into the effectiveness of SIMD in improving algorithm performance.

Task 2.2. A review on implemented algorithms. In addition to reviewing papers, we will also gather information on software that is known to provide efficient implementations of DATA and

DNA processing tools accelerated by SIMD-code. While it is not possible to compile a complete list of such software, we will focus on identifying sources of high-quality code, including high-performance libraries (such as Hyperscan or Intel's SIMD sorting library), standard or widely used libraries of popular programming languages (such as glibc, boost, and bitmagic), and databases-related tasks (such as regex execution engines, SIMDJson, XML parsing, and CSV parsing). From DNA-processing, we will harvest state of the art implementations in DNA-alignment as for instance ksw2 or bwa-mem2.

Task 2.3. Structuring the knowledge. One goal of this review process will be to coalesce a *collection of examples*, in a normalized format that is easy to distribute to other communities (for instance, using C). This collection will be compiled and normalized manually, completed with descriptions and sources, along with test dataset for the examples and performance measurements (obtained via Grid5000). The aim of this collection is twofold: serve as a *performance* benchmark for compiler design and optimisations, including our own work on VIR; and serve as an *expressivity* goal for language designers with simple and complex SIMD programs for streaming applications and their test inputs. Such set of programs will be essential to design Vectoid and ensure its supports of features required in practical applications. The set of examples will be publicly available online, and published as a technical report detailing the content of this WP. We will also present the interesting examples at the second project meeting. Ultimately, we will submit a complete survey for publication.

WP 3. Development of Vectoid.

Participants: G. Radanne, L. Henrio, M. Moy, D. Kuperberg, PhD2, Engineer1 (Lyon); C. Paperman, M. Hauspie, S. Salvati (Lille)

This WP regroups all tasks related to the design, formalization and implementation of Vectoid, to be explored conjointly with VIR compilation (WP4) and formal study of vectorial circuits (WP5). We will first extend the prototype developed in WP1 with challenging features, and develop two implementations: a simple sequential one, used to reason formally about the language, and the full-blown optimizing compilation setup using VIR. To make all this work easily usable, we will fine-tune its integration with Rust. Finally, we will evaluate our work with case studies taken from WP2.

This WP organization will be centered around the LIP center and the CASH team while D. Kuperberg will help with the interaction between WP3 and WP5. A PhD will be recruited to work on tasks 3.1 and 3.3. An engineer will be hired for the last year, specifically on task 3.4 and 3.5. The CRIStAL team will contribute to all aspects of the WP.

Task 3.1. Language definition. From a language perspective, there are three main difficulties to tackle: conditionals, functions, and synchronisation.

As highlighted in Section a.1, vectorial programs handle some form of branching extremely efficiently, but are limited in their general expressivity. Providing expressive constructs in Vectoid is essential for ease-of-programming, but we still want to stay in the program fragment that results in high performances. For this purpose, we aim to combine careful language restrictions (either syntactic or using static analysis), our knowledge of the expressivity of vectorial circuit, and existing compiler transformations such as if-conversion, in link with the design and compilation of VIR.

Modularity of high-performance code, and SIMD-code in particular, is difficult: while providing function calls would be the minimum syntax design to enable modular development, repeated function calls within the main execution loop can degrade performance greatly. A possible approach to avoid this pitfall is to aggressively inline functions. Investigating more sophisticated methodologies, such as compiling partially evaluated functions or allowing a limited amount of function calls, will be challenging but worthwhile.

A natural functional operator is `filter`, which filters the element of a stream and returns a sparser stream, unsynchronized from the input one. Such operator can be implemented using existing vectorial instructions that prune unwanted byte (see vcompressb in AVX-512), however, this now requires providing ways to combine and manipulate desynchronized streams. The *Synchronous languages*

community have developed analyses and type systems to model the clock of signals, which we could reuse for our purposes, with a more dedicated goal towards performances.

Task 3.2. A sequential engine. While our end goal is an optimizing compiler, reasoning about programs by translation to a lower-level representation such as VIR is difficult. To provide an easier-to-grasp mental model for developers, we will develop formal semantics for the language and develop a simple compiler that produces purely sequential code, without considering SIMD optimization. While we do not anticipate any scientific challenges, this task will establish a benchmark and a ground truth for further optimization efforts in the SIMD world, by allowing for comparative testing with the more optimized implementations.

Task 3.3. Design and compilation to VIR. VIR should precisely capture the expressivity of vectorial programs. However, Vectoid is a much higher level language, and the initial step of compilation from Vectoid to VIR will be non-trivial and designed conjointly with VIR. This task will contain the handling of conditionals and functions presented before, along with code simplification specific to our functional setting (for instance, by leveraging immutability [18]). Crucially, this part is decoupled from the *back-end* compilation of VIR, which will handle architecture-specific SIMD instructions and optimizations inspired by theoretical advances in vectorial circuits. This allows us to focus on high-level transformation and simplify the design of the compiler and the language.

Task 3.4. The interplay with the host language. When creating a domain-specific language, it is common to deal with the interaction with generic-purpose host languages only at the end of the project, as part of the packaging process. In our case however, any significant overhead between the Vectoid code and the host language could ruin performances, making our work worthless. In particular, when returning whole streams, or complex structures, we want the cost of interaction to be minimal.

Additionally, the interplay with system libraries will raise other challenges. While running on fixed buffer of memory should be rather easy, providing a dedicated and efficient interface of Vectoid with streaming ability of systems (e.g. with network stack or with system pipes) will be challenging.

For these reasons, we will study interactions between Vectoid and Rust right from the beginning of the project and ensure that they raise no additional cost by leveraging existing work on efficient streams in Rust. Furthermore, we will provide effective integration with standard libraries from the ecosystem, notably for IO and asynchronous programming. We will also package our work as a Rust crate and distribute it in the Rust package repository.

Task 3.5. Expressivity, benchmarking, and applications. Finally, to assess that our work is actually useful for SIMD programmers, we will evaluate Vectoid on the example set produced in WP2, along with benchmarks of handcrafted vectorial code.

Special attention will be given to the benchmarking setup and methodology. Reproducibility of the benchmarks will be ensured through careful handling of benchmarking data, versioning of the benchmarked code and archiving of the dataset on a publicly accessible repository.

Additionally, this task will explore applications, depending on the progress made on the language and compiler. Some string manipulation libraries could be improved using Vectoid. We also believe that most of the coreutils tools [53], such as `cut`, `tr`, `wc`, etc., can be built with the first iterations of the language. These proof-of-concept applications will be publicly advertised and will form the basis of the future tutorial to be written in WP6.

WP 4. Vectorized circuit compilation.

Participants: L. Gonnord, B. Ferres, V. Egloff (Grenoble); C. Paperman, M. Hauspie, S. Salvati, PhD1 (Lille); G. Radanne, M. Moy, PhD2 (Lyon); N. Fijalkow (Bordeaux),

In this WP, we explore the design of VIR, its compilation to various back-end architectures with SIMD instructions, its optimization. It will be managed by Laure Gonnord at LCIS. As it will require a lot of synchronization with WP3 and WP5, both PhD students and their (co-)advisers will participate.

Task 4.1. VIR design. The first challenge of this WP will be to design an IR that is expressive, minimal, and accommodate multiple targets. This design is heavily dependent on both WP3 and WP5. Carefully designing VIR's expressivity is essential to make it a compilation target for Vectoid but also other calculus (e.g. regexp or query languages). Isolating the minimal set of operations in VIR will open up possibilities for specific VIR transformations, optimizations, and formal analysis. Finally, the *multi-target* approach is required by the high diversity of SIMD instruction sets.

The design of VIR, will take inspiration from two sources: the theory of vectorial circuits on one hand (WP5), which will dictate its expressivity and some dedicated optimisations; and synchronous languages such as Lustre on the other hand, for their handling of streams and synchronisation. The compilation of synchronous languages has already partially addressed the *data to control flow* problem, although with a very different end goal (critical systems initially, with no parallelism at all), providing us with initial designs on which to expand with SIMD-specific considerations. Among synchronous languages features, we would take inspiration of clock calculus (to infer synchronisations), array iterators [29], multi-clock and multi-rate variants [35], and multi-task and parallel code generation [44].

Task 4.2. VIR implementation. As a potential implementation support, we will study the feasibility of using MLIR [26], the novel intermediate representation inside LLVM. Using it provides some major benefits: it embeds many already existing code optimisations, and makes use of a hardware independent vectorial dialect. While it is not well suited by design for dataflow processing, it has been proposed recently in a dataflow setting [7]

Then, after carefully designing the appropriate compiler infrastructure, we will build a complete compiler stack for Vectoid based on this choice and on the experience built in WP1. One first step will be a working PoC that exhibits several mappings from VIR high level operation to backend-dependent SIMD-code. These mappings being crucial for the further development of the compiler, a specific attention will be devoted to their specifications and it will be the topic of a dedicated technical report. A concrete evaluation of the tool-chain with at least two diverse backends will be done at this step.

Task 4.3. VIR to VIR. The code transformation performed when turning VIR into SIMD code will make the code quite convoluted, obscuring the semantics to the back-end compiler. Hence we will make some transformations directly on the VIR (constant propagation, compile time evaluation), before generating sequential (vectorized) code. Crucially, we will also develop VIR-specific transformations inspired by the study of vectorial circuits and dataflow programs. As an example, subsequent map operations can be fused, such also fusing the SIMD operations they are compiled to. After a precise evaluation of code generation and optimisations, we will stabilize the proof of concept and propose a complete release with improved code generation.

Task 4.4. RISC-V "V". *RISC-V* is an open-source instruction set architecture that is currently under rapid development, with increasing numbers of implementations. It is based on a set of extensions and with extensibility capabilities so that user defined extensions are easily added. Recently, a task force within the RISC-V project created a vectorial extension set for RISC-V called "V" [1]. Although no hardware has run these instructions yet, it is already possible to develop a compiler targeting this instruction set. Our goal will be to try to compile VIR to RISC-V "V". Targeting such architecture presents two main interests, beside the growing popularity of the RISC-V ISA in the hardware community. First of all, as the instruction set is open-source, our experimentations on compiling data-processing application to RISC-V "V" will be used to evaluate the possibility itself of doing so, and will result either on a successful backend, or in an interesting feedback for the standardization committee. Depending on our result, we might require some more instructions to be added to the instruction set, and RISC-V is the ideal project for this. Moreover, the RISC-V "V" extension present an interesting (and unique) characteristic, as it offers dynamic vector sizing (using configuration registers inside the vectorial architecture). It will hence be interesting to explore the benefits of such feature in our compiler. If time and resources allow, we may evaluate the output of our backend using simulations and/or benchmarks on FPGA targets.

WP 5. Formal methods for VIR.

Participants: C. Paperman, S. Salvati, M. Hauspie, PhD1 (Lille); D. Kuperberg (Lyon); L. Gonnord (Grenoble), N. Fijalkow (Bordeaux)

The goal of this work package is to construct mathematical tools to gain insight and feed the development of VIR and Vectoid. In particular, we build on methodologies from formal methods and automata theory. This WP will be centered around CRIStAL (Lille) with interactions with LIP (Lyon) and LaBRI (Bordeaux) through their expertise in logic and automata theory. The PhD student (PhD2) will contribute to tasks 5.1 and 5.2 of this WP with a specific attention on their integration to WP4. For this reason, L. Gonnord will also contribute to this WP.

Task 5.1. Formal expressivity of VIR and its efficient fragments. This task aims to understand the expressive power of VIR from an automata-theoretic and formal language perspective. To do so, we will create a sequential model of computation allowing formal reasoning on the capabilities of VIR. In terms of classical models of computation we don't expect VIR to have a standard expressivity: well-matched parentheses are not expressible as they require memory managements, while counters are available. Hence, the model is orthogonal to the classical Chomsky hierarchy. This investigation will be extended to the efficient hardware-related fragments of VIR. Through this exploration, we hope to gain insights into the nuances of VIR and their performance on specific hardware. The final goal is to design decision procedure for those fragments and to use them in the compilation stack.

Task 5.2. Understand the relationship between circuit complexity and VIR. VIR circuits can be seen as a *uniform description* of families of Boolean circuits and therefore naturally embed into circuit complexity classes (see [20]). The zoology of circuit complexity has been the topic of intensive theoretical research for the last fifty years. The goal of the task is to connect the dots between VIR and circuit complexity and to harvest from circuit complexity lower bounds methodology and approaches for proving inexpressibility results. By relying on the well known complexity of regular languages [45], we believe it will be possible to find *small automata* without any possible compilation procedure in VIR and hence, showcase formally proven hard-case to vectorize.

Task 5.3. Precondition on the input stream. Data-processing programs do not go through arbitrary streams but on streams with some inner structures, often known to the developer. In traditional programming language, such invariant are often difficult to convey to compilers, leading to missing compilation opportunities and significant manual, and often brittle, optimisations. In link with Vectoid's design, we plan to allow labelling input stream with logical constraint and leveraging these specifications in the compilation process. This endeavor is related to *separation problems*, which is already equipped with topological and algebraic tools that we hope to utilize. In programming language design, many features have been developed to encode semantic invariant in types [49]. We can revisit these approaches, which are generally focused on safety, for the purpose of efficiency.

Task 5.4. Relationship between query languages and VIR. One of the practical goal to VIR is to simplify data-processing efficient code generation. In this task we will focus on two data-processing applications: regular expression matching and XPath-like querying of streamed-trees. Both are the topic of intensive academic and industrial work, including projects taking advantage of SIMD for performance gains. As illustrated by the previous task, we believe that VIR will not be able to capture all finite automata. Hence, it is possible to try to identify fragments of regular languages of words and trees that can be computed in VIR. Once those fragments are well-identified in theory, we will develop dedicated compilation to VIR and integrate them into regular expression engine and into the project rsonpath. We will study other potential applications on related topics (XML processing and CSV processing) and use them as case studies of the usability and efficiency of Vectoid.

Task 5.5. Application dedicated to security. Many security oriented applications heavily rely on fast data-processing and could benefit from the methodologies developed in the WP, in particular from the last task. Typical examples would be network Firewalls, Intrusion Detection Systems, malware detection programs or even routers or switches in Software Defined Networks (SDN). Among

other techniques, these applications use signatures-based algorithms or clustering/membership check algorithms to perform their duty. For these applications, the algorithms produced must keep pace with the ever growing network speed and respond in near real time. Signatures based algorithm usually use either regular expressions or hash based signatures that can leverage vectorization. Membership checking or clustering – such as Bloom filters – can also benefit from vectorization [28]. Thus, we will investigate if the application of Vectoid would help to produce efficient code for those use cases.

This task will study security-oriented applications (such as Snort or Suricata in the IDS field or ClamAV for anti-malware solutions) and analyse if, and how, those applications fit in Vectoid.

WP 6. Application to DNA-processing.

Participants: A. Limasset, C. Paperman, Engineer2 (Lille); G. Radanne (Lyon)

Pairwise sequence alignment is a critical task in sequence Bioinformatics. It most prominent industrial application is to match *reads* (small chunks of DNA outputted by a sequencing machine) to reference genome. With sequencers now capable of producing terabytes of data per day and the *sequence read archive* containing over 36 petabytes of sequences, optimizing algorithms for indexing, searching, and exploring this vast amount of data is crucial. However, due to the variety of tasks requiring different types of distances (edit, hamming, gap-linear, gap-affine), different goal (local, semi, global) and different type of scores, the problem of sequence alignment is multifaceted and requires a large number of algorithms optimized for different requirements. Adapting these algorithms for novel hardware architectures is an ongoing and resource-intensive task at the detriment of more fundamental research. This is where Vectoid comes in. By providing a high-level programming language for SIMD processing, Vectoid has the potential to simplify the development of optimized tools and reduce the engineering effort required for adapting these algorithms to new hardware architectures.

This WP will be centered around A. Limasset, a leading expert in efficient DNA-processing algorithms. To ensure the success of the WP, team leaders in LIP and CRIStAL will be involved and a dedicated engineers (Eng2) will be recruited within CRIStAL.

Task 6.1. Vectoid and Bioinformatics. This task summarizes a two way collaboration: using Vectoid for Bioinformatics, and take input from Bioinformatics users to improve Vectoid. First, we will write several examples of SIMD-optimized DNA-alignment algorithms with Vectoid. Second, we will use this feedback to adjust and improve Vectoid for Bioinformatics tasks. Through this study, we hope to demonstrate the utility and effectiveness of Vectoid for Bioinformatics and highlight its usefulness in performance critical software engineering efforts. We also aim to publish this use of Vectoid to encourage other Bioinformatics researchers to try and adopt Vectoid. The targeted outcome of applying Vectoid in this context is to drastically lower the time-consuming task of applying SIMD-optimizations to the overwhelming amount of variation around DNA-alignment algorithms.

Task 6.2. Improving local sensitive hashing algorithms. A novel trend in Bioinformatics to compare long sequences is to transform them into smaller sketches inspired by *local sensitive hashing* methodology. Efficiently comparing, indexing and compressing such sketches adapted to DNA sequences is a very recent topic, well in scope of Antoine Limasset's research [37]. Vectorization of those hashing algorithm will have a massive impact on their performance but would requires intricate programming that Vectoid ought to gracefully support.

WP 7. Consolidation and diffusion.

Participants: C. Paperman (Lille); G. Radanne, Engineer1 (Lyon)

This Work Package is dedicated to ensuring the successful conclusion of the project. Its objectives are to advance the maturity level of Vectoid and make it easily usable for the general public, as well as to synthesize the project's findings and insights into a comprehensive paper. This WP will involve the team leader in CRIStAL (Lille) and LIP (Lyon) as well as a recruited engineer (Eng1) at LIP.

Task 7.1. Vectoid for the general public. To reach this level of maturity, several essential tasks must be completed. These include writing documentation, creating clear and illustrative examples, testing the code thoroughly, and preparing tutorials to help developers get comfortable with the syntax and limitations of Vectoid. Detailed documentation is crucial in providing an overview of Vectoid's abilities, making it easier for users to integrate the tool into their projects. Creating examples is essential as it demonstrates Vectoid's applicability in real-world scenarios, making it easier for potential users to understand and integrate it in their project. Thorough testing is essential to identify and rectify errors before the tool is released, ensuring that Vectoid performs as expected. Lastly, preparing tutorials is critical in helping developers to understand how to use Vectoid and its various features effectively. The completion of these tasks is essential in creating a high-quality software library that will serve as a solid foundation for Vectoid's future development and evolution.

Task 7.2. Building a consortium. To ensure the sustainability of the project in the future, particular attention will be paid o the creation of a consortium of Vectoid users. The aim of the consortium is to ensure the sustainability of the project by driving its evolution and further development. All members of the **Shannon meets Cray** project will be included into those discussions and the user-base will be chosen amongst the {data, DNA}-processing researchers and industries.

Task 7.3. *Lessons learned* paper. To effectively conclude the project scientifically, this WP team will prepare a *lessons learned paper*, which will be submitted for publication. This paper will synthesize the knowledge and insights gained during the development of Vectoid and all the lessons learned about generating efficient SIMD-code. This paper will hopefully help to better understand SIMD-programming and could become an important step for researchers and practitioners interested in SIMD-programming and vectorizing. It will provide valuable insights that can help improve future generations of compilers and auto-vectorizers.

WP 0. Risk management.

We discuss a number of risks we have identified and concrete ways to mitigate them: scientific hurdles, timing, recruitment, code efficiency, theoretical developments, and expertise. Our goal for Vectoid is ambitious as it discusses a number of advanced features which would have groundbreaking implications in the field of compilation, and towards applications for data and DNA processing. Although we did carefully consider and calibrate each of the features we plan on implementing in Vectoid, there may be insurmountable difficulties. To address this issue, the project is built in a non-sequential manner: even if some features cannot be implemented, it does not impede the project as a whole. All phases of the project can be suitably adapted to account for lack of success in a concrete task. The project is planned for five years and includes tight timing constraints aiming at reaching a complete implementation of Vectoid. In particular, the preliminary phase is crucial and must be realised within the timing since it conditions the rest of the project. To ensure the success of the preliminary phase, all developments will be done in Lille and all people involved in this phase have already been working together harmoniously.

Recruitment. To realise our project we need a broad set of skills and therefore plan on hiring a number of people in the project, in particular junior (interns, PhD students). We are aware of the difficulty in hiring students with the target background. As evidence that we will find suitable candidates, we have in the past found excellent students who greatly contributed to the maturation of the current proposal, and finished their studies in prestigious universities (ENS, grandes écoles, University of Warsaw). A crucial aspect of the project is to obtain an industrial-grade implementation of Vectoid, which is essential for the applications we envision. For this reason, an important part of the implementation will be carried out by software engineers, with support from INRIA engineers.

Theoretical developments. We see the theoretical developments towards understanding the expressive power of Vectoid as a backbone of the project, which will drive our implementation effort. Since the model is non-standard, the usual tools from automata theory, logic, and circuit complexity

may not be easily adapted. We believe that the combination of the expertise of the theoretically-inclined researchers of the team gives a reasonably high chance that theoretical developments will indeed bring important insights towards materialising Vectoid.

Expertise. The project is by essence interdisciplinary, as it involves very different fields and a wide range of expertise: rooted in compilation, it aims at involving theoretical developments from automata theory, logic, circuit complexity, and aims at applications in data and DNA processing. We believe that we have assembled a very strong team of researchers with the required expertise. Many of the researchers in the project have already worked together in the past, which is evidence that collaboration at the scale of this ambitious project is feasible.

d Ability of the project to address the research issues covered by the chosen research theme

This research project aims to tackle a long standing problem in compilation (vectorization) by relying on methodology from programming language design, logic, automata theory and circuits complexity. In terms of methodology, the project is included into the Axe E.3 (keywords **langages de programmation** ; **modèles de calcul pour le parallélisme**). Hence, this project is submitted to the *comité d'évaluation scientifique 25: Sciences et génie du logiciel - Réseaux de communication multi-usages, infrastructures numériques*. However, this project includes research that would also fit the Axis E.1 (keywords **informatique fondamentale** ; **langages et sémantiques** ; **logique**; **méthodes formelles** ; **modèles de calcul**), given its root in formal methods. Finally, a part of the project is dedicated to applications to DNA processing which is included into the Axis H.14 *Interfaces : mathématiques, sciences du numérique – biologie, santé* and in particular with the keywords **big data en biologie** and **bioinformatique**.

II Organisation and implementation of the project

The actors of the partnership are CRIStAL (INRIA-university of Lille), the LIP (INRIA-ENS Lyon), the LCIS (Grenoble INP) and LaBRI (Bordeaux), with members covering its diverse themes from theoretical computer science, compilation, architecture design, data-processing to DNA-processing. See Table 3 for a summary of the implication of each center leaders in on-going projects.

a Scientific coordinator and consortium

Lille (CRIStAL). Charles Paperman, PI of the project, is an expert in logic in computer science, with a focus on circuit complexity of automata; and more recently on circuit complexity, vectorization and efficient data-processing. Sylvain Salvati is expert in logic, semantics and programming languages. Michael Hauspie is expert in domain specific languages and low-level optimization in resource constrained systems. They all focused recently on compilation of automata to SIMD as part of the PhD

Name	p.m	Call	Title	PI's Name	Dates
C. Paperman	5	Region HdF	NetworkDisk	C. Paperman	10/22–12/23
N. Fijalkow	45	ANR JCJC	G4S	N. Fijalkow	01/22–09/26
L. Gonnord	3	PEPR	ARSENE	CEA	06/22–06/27
L. Gonnord	2	AMI	CMA	V. Quéma	06/23–06/28
L. Gonnord	5	AMI	CMA Cyberskills	C. Pernet & L. Gonnord	06/23–06/28

Table 3: Implication of the scientific coordinator and partner's scientific leader in on-going project(s)

thesis of Claire Soyez-Martin [36]. Antoine Limasset, a junior CNRS researcher in the Bonsai team, is a specialist in high performance computing for DNA processing.
Lyon (LIP). Gabriel Radanne is an expert in domain specific languages, high level languages, and their compilation. He has already designed and implemented languages dedicated to a number of domains, along with optimizing compilers for dedicated embedded languages. Members of the CASH Team (Ludovic Henrio and Matthieu Moy) will contribute with their expertise in semantics, parallelism, and low level architectures. Denis Kuperberg is an expert in logic and automata theory and will help link the automata and compilation approaches.
Grenoble INP (LCIS/VERIMAG). Laure Gonnord (formerly in LIP) is an expert in synchronous compilation, parallelism, static analysis and will benefit of the LCIS/VERIMAG hardware architects expertise (Valentin Egloff and Bruno Ferres).
Bordeaux (LaBRI). Nathanaël Fijalkow is an expert in program synthesis, combining machine learning and formal methods. He will contribute to compilations, optimisation and language design.

b Implemented and requested resources to reach the objectives

Recruitment within the project. Two PhD students will be hired in the main research phase of the project. The first PhD student (PhD1) will be in CRIStAL co-advised by Charles Paperman and Laure Gonnord (LCIS); he will work on the relationship between VIR and vectorial circuits and its implication toward VIR compilation (WP4 and WP5). The second PhD student (PhD2) will be located at LIP, he will be advised by Gabriel Radanne. He will work on the design of Vectoid and its compilation to VIR (WP3). The completion phase will be supported by two engineers: one (Eng1) to help consolidate Vectoid development (integration, tests, documentation) at LIP for two years and one (Eng2) in CRIStAL to apply Vectoid to DNA-processing for one year. For those recruitment, a special attention will be devoted to gender representation through active solicitation to candidates. Some internships (two in each center) will be offered during the project. The first interns will participate in the early development phase, especially of the Vectoid PoC, and be potential PhD students. They will be hired from the best Universities and Grandes Écoles using our research network. The latter interns will be engineer-oriented students who will participate in the dissemination and application efforts, in particular towards data and DNA-processing.

The *operating cost* allocation will be used to organize periodic team meetings, work package meetings, and facilitate visits among consortium members. The consortium will meet six times, twice in Lille and Lyon, and once in Bordeaux and Grenoble. We anticipate that the cost of each meeting will be approximately 5000€. Remaining operating costs will be allocated for work package-specific meetings and research visits. Additionally, we have budgeted participation of PhD students to summer schools and regular visits from PhD1 at Grenoble to meet with Laure Gonnord, who will be co-supervising the PhD thesis (part of Lille's budget). The budget is recapitulated in Table 4.

	CRIStAL	**LIP**	**LCIS**	**LaBRI**
Staff expenses	178 000€	225 600€	8 400€	7 800€
Material costs	3 000€	2 000€	1 000€	1 600€
Operating costs	25 000€	20 000€	6 000€	10 000€
Management fees	29 870€	35 902€	2 233€	2 813€
Sub-total	**235 870€**	**283 502€**	**17 633€**	**22 213€**
Requested funding	**559218€**			

Table 4: Budget summary of the **Shannon meets Cray** project

III Impact and benefits of the project

The main outcome of the project is to release the first domain-specific language for designing efficient streaming SIMD-programs. Its adoption within the data and DNA processing communities will demonstrate the success of the **Shannon meets Cray** project. By providing a specialized language, it aims to significantly reduce the entry-cost for producing efficient streaming algorithm implementations, as well as improve their maintainability and portability in the long run. Efficient software implementation is becoming increasingly crucial with the growing number of fields generating exascale amounts of data. Although producing efficient algorithms is not an objective *per se*, the large number of research papers published each year on SIMD-implementation highlights the scientific community's need to create more efficient software. The development of such efficient code will open new avenues for applications that can cope more easily with high throughput data at a lower cost.

From a computer science perspective, vectorization remains a challenging area. Although some compilation techniques exist in specific settings, auto-vectorization is still an untamed beast. We believe that fundamental reasons are yet to be understood to explain the difficulties in producing efficient SIMD-code. The relationship between circuit complexity and vectorial algorithms will be a key factor in understanding these challenges and the methodology introduced in this project will help advance our understanding of these questions. In the long run, we hope that compilers will make Vectoid obsolete by integrating powerful auto-vectorization compilation procedures, but to reach this goal Vectoid is a necessary intermediate step. Furthermore, by proposing Vectoid, the underlying disciplinary goal of the project is to understand the interplay between parallelism and sequentiality. This interplay is much broader than the scope of the project and has implications way beyond SIMD (e.g. FPGA, ASIC or *processing in memory*).

IV References related to the project

[1] RISC-V V vector extension.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. POPL, 1983.

[3] C. Barloy, F. Murlak, and C. Paperman. Stackless Processing of Streamed Trees. In *PODS*, 2021.

[4] A. Benveniste, P. L. Guernic, and P. Aubry. Compositionality in Dataflow Synchronous Languages: Specification and Code Generation. In *The Composer*, 1997.

[5] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[6] G. E. Blelloch, J.C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a Portable Nested Data-Parallel Language. *Journal of Parallel and Distributed Computing*, 1994.

[7] Pedro Ciambra, Mickaël Dardaillon, Maxime Pelcat, and Hervé Yviquel. Co-optimizing Dataflow Graphs and Actors with MLIR. In *SiPS*, 2022.

[8] J. Daily. Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC bioinformatics*, 2016.

[9] P. Feautrier. Array expansion. In *ICS*, 1988.

[10] M. Furst, J. B. Saxe, and M. Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 1984.

[11] M. Gienieczko, C. Paperman, and F. Murlak. rsonpath, SIMD-powered JSONPath, 2022.

[12] N. B. B. Grathwohl, F. Henglein, U. T. Rasmussen, K. A. Søholm, and S. P. Tørholm. Kleenex: compiling nondeterministic transducers to deterministic streaming transducers. In *POPL*, 2016.

[13] T. Grosser, A. Größlinger, and C. Lengauer. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Process. Lett.*, 2012.

[14] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet. Polly-Polyhedral optimization in LLVM. In *IMPACT*, 2011.

[15] T. Gubner and P. Boncz. Exploring query execution strategies for JIT, vectorization and SIMD. *ADMS@VLDB*, 2017.

[16] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 1991.

[17] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector SIMD architectures. In *ICS*. ACM, 2013.

[18] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *PLDI*, 2017.

[19] LLVM Vectorization Plan.
[20] N. Immerman. Descriptive Complexity. 1998.
[21] Intel. Guidelines for Writing Vectorizable Code.
[22] L. Jiang and Z. Zhao. JSONSki: streaming semi-structured data with bit-parallel fast-forwarding. In *ASPLOS*, 2022.
[23] L. Lamport. Multiple Byte Processing with Full-Word Instructions. *Commun. ACM*, 1975.
[24] G. Langdale and D. Lemire. Parsing gigabytes of JSON per second. *The VLDB Journal*, 2019.
[25] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 2012.
[26] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *CGO*, 2021.
[27] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In *HPCA*, 2012.
[28] J. Lu, Y. Wan, Y. Li, C. Zhang, H. Dai, Y. Wang, G. Zhang, and B. Liu. Ultra-Fast Bloom Filters using SIMD Techniques. *TPDS*, 2019.
[29] L. Morel. Efficient compilation of array iterators for Lustre. In *Synchronous Languages and Applications*, 2002.
[30] F. Murlak, C. Paperman, and M. Pilipczuk. Schema Validation via Streaming Circuits. In *PODS*, 2016.
[31] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, 1999.
[32] D. Nuzman, S. Dyshel, E.. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks. Vapor SIMD: Auto-vectorize once, run everywhere. In *CGO*, 2011.
[33] D. Nuzman and A. Zaks. Autovectorization in GCC–two years later. In *GCC Developers Summit*, 2006.
[34] D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short simd architectures. In *PACT*, 2008.
[35] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 2011.
[36] C. Paperman, S. Salvati, and C. Soyez-Martin. An Algebraic Approach to Vectorial Programs. In *STACS*, 2023.
[37] G. E. Pibiri, Y. Shibuya, and A. Limasset. Locality-Preserving Minimal Perfect Hashing of k-mers. In *ISMB 2023*.
[38] V. R. Pratt, M. O. Rabin, and L. J. Stockmeyer. A Characterization of the Power of Vector Machines. STOC, 1974.
[39] M. Rainey, R. R. Newton, K. Hale, N. Hardavellas, S. Campanoni, P. Dinda, and U. A. Acar. Task Parallel Assembly Language for Uncompromising Parallelism. PLDI 2021, 2021.
[40] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 1978.
[41] O. Serre. Vectorial languages and linear temporal logic. *Theoretical Computer Science*, 2004.
[42] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Electrical Engineering*, 1938.
[43] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 1981.
[44] J. Souyris, K. Didier, D. Potop-Butucaru, G. Iooss, T. Bourke, A. Cohen, and M. Pouzet. Automatic Parallelization from Lustre Models in Avionics. In *ERTS2*, 2018.
[45] H. Straubing. Finite automata, formal logic, and circuit complexity, 1994.
[46] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. *ACM SIGPLAN Notices*, 2006.
[47] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *CC 2002*. Springer, 2002.
[48] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *PACT*, 2009.
[49] N. Vazou, E. L. Seidel, and R. Jhala. LiquidHaskell: experience with refinement types in the real world. In *ICFP*, 2014.
[50] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *NSDI*, 2019.
[51] Wikipedia. CLMUL instruction set.
[52] Wikipedia. Cray-1.
[53] Wikipedia. GNU Core Utilities.
[54] Haoran Xu and Fredrik Kjolstad. Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode. *OOPSLA*, 2021.