CORAL: Verification-aware OpenCL based Read Mapper for Heterogeneous Systems

Sidharth Maheshwari, Venkateshwarlu Y. Gudur, Rishad Shafik, *Senion Member, IEEE,* Ian Wilson, Alex Yakovlev, *Fellow, IEEE,* and Amit Acharyya, *Member, IEEE*

Abstract—Genomics has the potential to transform medicine from reactive to a personalized, predictive, preventive and participatory (P4) form. Being a Big Data application with continuously increasing rate of data production, the computational costs of genomics have become a daunting challenge. Most modern computing systems are heterogeneous consisting of various combinations of computing resources, such as CPUs, GPUs and FPGAs. They require platform-specific software and languages to program making their simultaneous operation challenging. Existing read mappers and analysis tools in the whole genome sequencing (WGS) pipeline do not scale for such heterogeneity. Additionally, the computational cost of mapping reads is high due to expensive dynamic programming based verification, where optimized implementations are already available. Thus, improvement in filtration techniques is needed to reduce verification overhead. To address the aforementioned limitations with regards to the mapping element of the WGS pipeline, we propose a **C**ross-platf**O**rm **R**ead m**A**pper using openc**L** (CORAL). CORAL is capable of executing on heterogeneous devices/platforms simultaneously. It can reduce computational time by suitably distributing the workload without any additional programming effort. We showcase this on a quadcore Intel CPU along with two Nvidia GTX 590 GPUs, distributing the workload judiciously to achieve up to 2× speedup compared to when only CPUs are used. To reduce the verification overhead, CORAL dynamically adapts k-mer length during filtration. We demonstrate competitive timings in comparison with other mappers using real and simulated reads. CORAL is available at: https://github.com/nclaes/CORAL

Index Terms—Genome, read alignment, OpenCL, heterogeneous systems, whole genome sequencing.

1 Introduction

Genomics will become a major generator of Big Data in the coming decade due to the advent of high-throughput sequencing and continued advances in the sequencing technology [1], [2]. The trickle-down effect has led to the establishment and expansion of the genomic data centers worldwide, to carry research in various fields including medicine, agriculture and forensics. Currently, over 6000 single-gene genetic conditions are known, and many other diseases involve genetic variants across the genome, making medicine the foremost application of genomics. Medicine is undergoing a transformation from its, largely, reactive nature to a personalized, predictive, preventive and participatory (P4) one [3]. Countries like the UK, Saudi Arabia, US and China are envisioning sequencing the genomes of a large part of their population [1] and working towards making genome sequencing and analysis a part of the routine tests performed at the hospitals. Genomic data, today, is largely generated in big sequencing centers or large commercial sequencing labs which generally employ state-of-the-art high-performance many-core systems made of monolithic hardware. In the P4 scenario, however, the installation and maintenance of such high-performance systems will be a bottleneck in setting up of cost-effective genomic healthcare infrastructure. Additionally, most modern platforms available are heterogeneous and consists of different computing hardware. Hence, whole genome sequencing (WGS) pipeline require platform independent computing framework to use all the available resources on the system for effective performance gains.

There are various categories of hardware computing resources such as central processing unit (CPU), graphical processing unit (GPU), field-programmable gate arrays (FPGA) and digital signal processors (DSP). These resources can be found in off-the-shelf platforms, provided by a range of electronics manufacturers such as Intel, AMD, Nvidia, ARM and Xilinx. They can be found either solo or in different combinations such as CPU + GPU, CPU + GPU + FPGA or CPU + FPGA. Most modern computing systems including many supercomputers, are heterogeneous and have a combination of CPU + GPU available on the same platform [4]. On the other hand, state-of-the-art bioinformatics tools, have focused on algorithmic innovations and software optimizations targeting, mainly, CPU [5], [6], [7], [8]. There are many communications available that have focused on acceleration of genomics algorithms on either GPU or FPGA, as summarized in [9]. However, to the best of our knowledge, a standalone tool capable of mapping reads using different devices, simultaneously, in a heterogeneous system is yet to be developed. It is arduous and challenging as these platforms have different architectures and, often, require vendor specific software and languages to program and use them. This will require rewriting or tailoring of the algorithms for portability.

- *S. Maheshwari, R. Shafik and A. Yakovlev are with the School of Engineering, Newcastle University, Newcastle Upon Tyne, UK, NE1 7RU. E-mail: s.maheshwari2@newcastle.ac.uk*
- *I. Wilson is with Institute of Genetic Medicine, Newcastle University, Newcastle Upon Tyne, UK, NE1 7RU.*
- *V. Y. Gudur and A. Accharyya are with Department of Electrical Engineering, Indian Institute of Technology (IIT), Hyderabad ,Telangana, 502285, India.*

Manuscript received January xx, xxxx; revised August xx, xxxx.

Read mappers	Cross-platform	Mapping locations reported			Preprocessing			Filtering		
		Best	All	First-n	Hashing	FM-Index	Suffix Array	Pigeonhole	q-gram lemma	Other
RazerS3			✓		✓			✓	✓	
Hobbes3			✓		✓			✓		
Yara			✓			✓	✓	✓		
FEM			✓		✓			✓		
GEM		✓				✓	✓	✓		
BWA-MEM		✓				✓	✓			✓
CORAL	✓			✓		✓	✓	✓		

TABLE 1
Characteristics of previously proposed read mappers along with our proposed CORAL. *Other* in the table implies a combination of multiple data structure, search algorithms and heuristics, the one, specifically, employed by BWA-MEM.

In this paper, we propose a **C**ross-platf**O**rm **R**ead m**A**pper using openc**L** (CORAL) to map reads on any OpenCL conformant device. Today, majority of platforms manufactured by different vendors comply with OpenCL standards [10]. We use OpenCL framework as a baseline to design the CORAL kernel and apply a series of algorithmic optimizations to subdue memory constraints. The aim is to enhance the portability of our aligner across various devices and platforms mitigating the need for restructuring or rewriting. CORAL is equipped to launch kernels, simultaneously, on all the available compute units, provided enough memory is available, to distribute the workload and achieve enhanced performance. With this feature, we address the limitations encountered in multi-device heterogeneous systems, ranging from servers to workstations, for the assembly element of the whole genome sequencing pipeline. The CORAL algorithm is fully sensitive and capable of reporting all mapping positions, however, the actual number of mappings reported is, mainly, limited due to the memory allocation restrictions imposed by OpenCL. We elaborate on this further in section 4.1. CORAL is verification-aware as it dynamically adapts the *k-mer*[1] length during filtration to reduce verification costs. It employs FM-Index [11] backward search to detect the number of candidate locations for a particular *k-mer* and in accordance with a threshold, it extends the *k-mer* to reduce the number of candidate locations to be verified. The candidate locations are obtained from suffix array data structure [12], preprocessed using reference genome, and are verified *in-situ* using banded Myers bit-vector algorithm [13], [14]. CORAL, automatically, determines the number of workitems (or threads) in a workgroup for a particular device based on user given workload allocation, distributes the workload and executes them in a task-parallel fashion. We write the host code in `Python` and kernel in `C` using OpenCL primitives. We use `PyOpenCL` rather than conventional `C`-based OpenCL as scripting language requires low programming effort. This, we believe, is an additional feature of CORAL as it enables fast modifications and prototyping.

Several investigations have been reported on performance and power-driven explorations of simultaneous kernel execution on CPU and GPU. The authors in [15] use OpenCL to run benchmarks, concurrently, on CPU and GPU cores of Odroid XU3 embedded platform. The goal is to exploit heterogeneity for better power-performance tradeoffs. They test it using Polybench benchmark suite [16] which contains general-purpose computing workloads. In [17], the authors present an investigation of simultaneous kernel execution on both CPU and GPU in a fused CPU-GPU architecture with shared LLC, mainly targeting the Intel platforms as they are the only one that, currently, supports OpenCL 2.0's fine grained SVM. They dynamically allocate the workitems, of Rodina benchmark suite, on devices to maximize performance. Dynamic work-item allocation is an ability provided in OpenCL 2.0 standard. In [18], the authors propose an energy-efficient run-time thread mapping and partitioning methodology for concurrent applications on Odroid XU3. All aforementioned works investigate the scheduling of kernels and partitioning of workitems between CPU and GPU on different platforms. They use workloads from standard benchmarking suites and attempt to improve performance-power tradeoffs. CORAL, on the other hand, is an application specific tool where we propose an algorithm for mapping reads efficiently and accurately across different platforms. It can be used on any OpenCL compatible device provided sufficient memory is available. We demonstrate, for the first time, a cross-platform read mapping scenario, which utilizes two Nvidia GTX 590 GPUs along with a quadcore Intel CPU to obtain better performance without any additional programming effort. We perform extensive validation and experimentation using both simulated and real reads and compare them with state-of-the-art read mappers.

2 Background

Read mappers can be classified as *all-mappers*, including RazerS3 [6], Yara [19], Hobbes3 [7] and FEM [8], and *best-mappers* including BWA-MEM [20], Bowtie2 [21] and GEM [5]. *All-mappers* attempt to identify all the mapping locations of the reads in the reference genome while *best-mappers* employ heuristics to identify the best mapping location. In general, *best-mappers* are faster than *all-mappers*, however, they may not report other mapping positions that are desired in downstream analyses and experiments including the ChIP-seq experiments, CNVs (copy number variation) calling and detecting structural variants. Aforementioned read mappers are based on read-alignment approach which assumes the availability of the reference genome. This approach has three methodological divisions: preprocessing, filtration and

1. *k-mer* is a subsection, of length *k*, of a read or genome.

verification, also known as, *seed-and-extend* in the case of *all-mappers*. The preprocessing stage uses data structures such as hashing, FM-Index [11] and suffix arrays [12] to store the reference genome. Filtration uses the preprocessed data structures and performs approximate string search of the reads with the reference genome. It prunes the reference genome using q-gram lemma or pigeonhole principle [2] to identify candidate locations where the reads may be located. The verification stage identifies the exact mapping location of the read. Most modern mappers employ banded Myers bit-vector algorithm [13], [14], which is a variant of semi-global dynamic programming. As verifications are expensive and runtimes of dynamic programming increase exponentially with the length of the strings, efficient filtration techniques are desired to narrow down the search space and reduce the total number of candidate locations per read. Every aligner, thus, attempts to extract appropriate *k-mer* from the read for low runtimes. [22] proposes one such technique, Optimal Seed Solver (*OSS*), to select the best possible *k-mers* so as to reduce the total number of candidate locations.

Table 1 shows the characteristics of previously proposed state-of-the-art read mappers. RazerS3 [6] uses hashing with open addressing to store the reference genome or the given reads and provides a choice of using either pigeonhole or q-gram lemma filter to map reads. It employs banded Myers bit-vector algorithm for verification. The tool achieves faster mapping times through user given sensitivity levels based on their proposed theory. Hobbes3 [7] uses pigeonhole filter along with a proposed novel dynamic programming based *k-mer* selection method to minimize the number of candidate locations, thereby, reducing the total verification time. FEM [8] is the latest mapping tool which constructs a succinct hashing index with low memory footprint to preprocess reference genome. It employs pigeonhole filter and uses *OSS* to select optimum *k-mer* lengths to minimize the total number of candidate location ensuring full sensitivity. GEM [5] is a *best-mapper* that uses FM-Index to preprocess the reference genome and employs heuristics to find the best matching position for a read. It uses adaptive *seeds* to reduce the total number of candidate locations during filtration and uses Myers bit-vector algorithm for verification. BWA-MEM [20] is a *best-mapper* that employs FM-Index, suffix arrays, dynamic programming based filtration and heuristics to report best mapping locations for the reads. Yara [19] is an *all-mapper* that uses pigeonhole along with approximate seeds to increase specificity of filtration and stores the reference genome using FM-Index and suffix arrays.

3 METHODS

In Section 3.1, we discuss OpenCL's unique way of looking at any hardware because of which CORAL can be used across different devices and platforms. It preprocesses the reference genome using FM-Index and suffix array. FM-Index offers flexibility in the variation of length of *k-mers* without any additional penalty, thus, accelerates search of any pattern in a text. The corresponding positions of occurrences can be found from suffix array. Details on the advantages of FM-Index are provided Section 3.3. Following that we elaborate on the preprocessing and filtration methodologies implemented by CORAL. For verification, we use banded Myers bit-vector algorithms whose details can be found in [6], [13], [14].

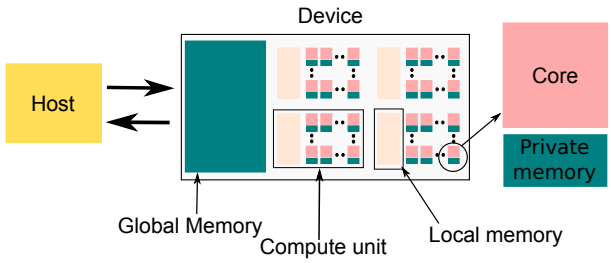

Fig. 1. OpenCL programming model with memory hierarchy.

3.1 OpenCL view of the hardware

Fig. 1 vizualizes how OpenCL [23] views a compatible hardware. From an execution standpoint, it recognizes two computational divisions viz. host and device; and three layers of memory with different access rights viz. global, local and private memory. Host is the master which issues instructions and data to the device for execution. The host and device communicate data through the global memory i.e. host cannot access the local and private memory of the device. The host and device need not be different platforms like a CPU-GPU pair or CPU-FPGA pair, it can be CPU-CPU pair where the host and device share the same global memory and compute resources in different time slots, meaning the host can launch kernels on itself along with other devices, if present. As we know, memory hierarchy in CPU consists of off-chip RAM and on-chip caches. Most CPUs have three levels of caches viz. level 1 (L1), level 2 (L2) and level 3 (L3), with increasing size and access times. These caches hold data which are accessed frequently, recently or both to improve the runtime of programs. GPU, in general, have "CPU-like" cores with multiple arithmetic logic units, cache and registers. Depending on the vendor, the internal architecture of the core varies along with the definition of GPU-core, however, all "CPU-like" cores in modern GPUs have registers and cache, generally, referred as L1 cache. OpenCL, generally, recognizes off-chip RAM and the L1 cache as global and local memory, respectively. Private memory, generally, are the registers available to the cores. Host issues instructions in the form of kernel, which are executed by workitems (or threads). workitems are equally divided into workgroups, with each workgroup occupying a single compute unit during execution. The workitems within a workgroup execute on all the available cores in the compute unit. As all compute units have separate L1 caches, all the workitems inside a workgroup share the local memory. The private memory, however, is only accessible to the workitem deployed on the core.

It is imperative to consider the memory capacities at all levels while designing the kernel. An efficient kernel minimizes private memory usage and the intra-data movements between private, global and local memory. There is no limit on the number of workgroups, however, the maximum number of workitems allowed in a workgroup depends on the device specification along with the private and local memory consumed by the kernel. Generally, GPUs

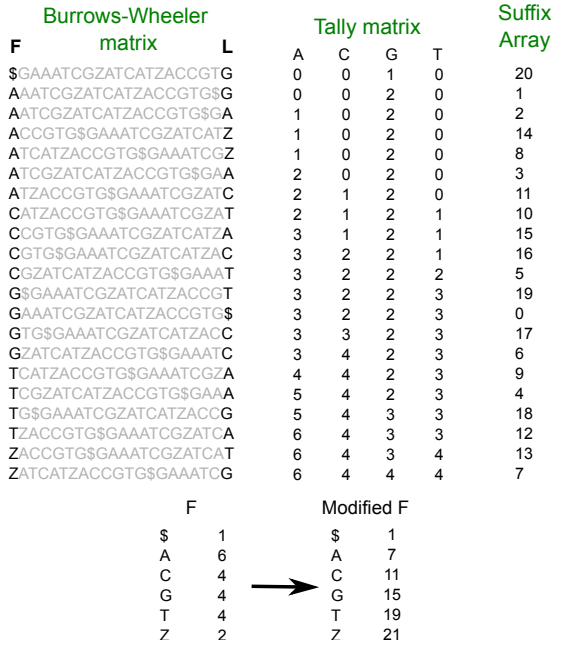

Fig. 2. Preprocessing methodology of CORAL for a small text: GAAATCGZATCATZACCGTG$ using FM-Index and suffix arrays. We store tally matrix, suffix array and modified **F** array to be used for querying *k-mers* in the filtration stage.

require low memory footprint kernels to achieve higher utilizations by engaging a greater number of workitems in a workgroup. Thus, designing of kernels with memory constraints require series of algorithmic optimization with respect to hardware. Our proposed CORAL kernel requires 380-480 bytes of private memory, depending on the read size, and does not use the local memory. Discarding the use of local memory in the kernel enhances the portability of CORAL as the size of local memory varies across different devices. Each workitem executes the kernel for a single read and performs the following operations: loading read in the private memory, identifying the candidate locations for both forward and reverse strand, performing *in situ* verification and writing the verification result back to global memory. As all reads are independent, asynchronous executions of the workitems result in better execution times.

3.2 Preprocessing

We use FM-Index and suffix array data structures to store the reference genome. FM-Index uses the first and last columns, denoted as **F** and **L**, of a matrix obtained by applying Burrows-Wheeler transform [24] on a string. This transform lexicographically sorts list of all reversible permutation of characters of a string, as shown in Fig. 2. Using the array **L**, we construct the tally matrix where each row stores the number of occurrences of alphabets starting from the first row up to and including a particular row, as shown in Fig. 2. Alongside, we also build a suffix array to indicate the position at which a particular alphabet from **L** occurs in the original string. **F** array can be compressed using run-length encoding to represent the total occurrences of all the alphabets in the string. We further modify **F** to provide cumulative numbers in the increasing order rather than the exact number of occurrences, as shown in bottom right of Fig. 2. The undetermined bases during sequencing of reads are represented as N. In our proposed preprocessing methodology, we replace N with Z as it occurs in the end of the lexicographical order of alphabets, thus, appearing in the end of **F** array, after bases A, C, G and T. For further details on FM-Index and suffix arrays, interested readers may refer to [11], [12], [25].

3.3 Verification-Aware Filtration

We demonstrate our pattern matching methodology, that employs FM-Index backward search, using an example where pattern ATC is searched in the text GAAATCGZATCATZACCGTG$, as shown in Fig. 3. We, also, show how search results can be extended if the pattern changes to AATC by prepending a character in the beginning, thus, increasing its size by one. Following that, we explain how pigeonhole principle along with verification-aware FM-Index backward search can reduce total number of candidate locations without affecting sensitivity.

3.3.1 FM-Index Backward Search

Before we proceed with FM-Index backward search, the concept of ranking of characters in the text needs to be explained. Rank of a character indicates the number of times that character has occurred in the text including the current instance. For example, the rank of bold character **A** in GAAATCGZ**A**TCATZACCGTG$ is 4. The purpose of the tally matrix obtained in Section 3.2 is to store the ranks of all the desired characters in the text. We do not store ranks for Z or N as they are considered errors and the corresponding *k-mer*, where it occurs, need not be searched.

Searching starts from the last character and moves up to the first taking as many cycles as the length of the pattern. Cycle 1 in Fig. 3 shows the number and positions of occurrence of C in the **F** array. We then look for the occurrence of next character i.e. T in the corresponding locations in **L** array and store the corresponding ranks from the tally matrix. We can see that T precedes C at two locations with ranks 1 and 2. Cycle 2 uses the, previously, stored ranks to locate the corresponding Ts in the **F** array. We, again, look for the next character i.e. A in the corresponding locations in **L** array and store the corresponding ranks from the tally matrix. At the end of cycle 2, we find that A precedes TC at two locations with ranks 4 and 5. Cycle 3, then, locates A in **F** array and reports the corresponding locations of occurrence from the suffix array. Fig. 3 shows that pattern ACT matches the text at positions 3 and 8 (zero based numbering). We can further continue searching if the pattern size is increased in a similar fashion using extended cycle 3 and cycle 4. At the end of cycle 4, the pattern AACT can found at location 2 with help of the suffix array. For all practical application, we do not need to store the **L** array rather the tally matrix, modified **F** array (Fig. 2), suffix array and the reference genome are required.

3.3.2 Verification-Aware filtration using Pigeonhole Principle

For mapping reads with an edit distance (or permissible error) of δ using pigeonhole principle, a read is divided into $(\delta + 1)$ non-overlapping *k-mers*. CORAL measures the

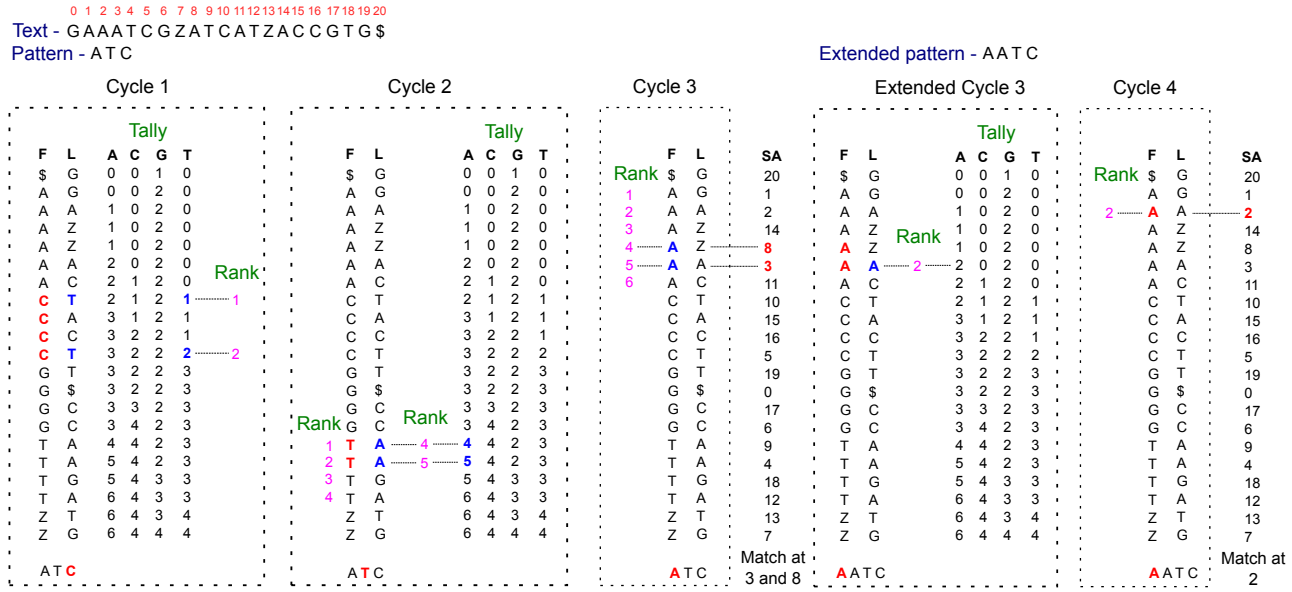

Fig. 3. Visualization of the searching method using FM-Index and suffix arrays. We search for pattern: ATC in the text: GAAATCGZATCATZACCGTG$ in three cycles and then show how the search can be extended if the pattern size increases on dynamically to AATC, using four cycles.

maximum possible k for equal length non-overlapping *k-mers*, as $k = \lfloor \frac{n}{\delta+1} \rfloor$, where n is the read length. For example, given $n = 100$ and $\delta = 5$, $k = \lfloor \frac{100}{6} \rfloor = 16$ and for $n = 150$ and $\delta = 7$, $k = \lfloor \frac{150}{8} \rfloor = 18$. We limit ourselves to $n = 100$ to 150 and $\delta = 0$ to 8 for reasons explained in Section 5.

Upon obtaining k, we can calculate the number of extra or spare bases (eb) that remain after securing $(\delta + 1)$ non-overlapping *k-mers*. For example, $eb = n - k \times (\delta + 1) = 4$ for $n = 100$, $\delta = 5$, and $eb = 6$ for $n = 150$ and $\delta = 7$. These extra bases can be used to extend the length of any *k-mer* as discussed in Fig. 3, to minimize the number of occurrences (or candidate locations) depending on the given threshold. It should be noted that the number of non-overlapping *k-mers* remain intact despite of extensions, thus, not affecting sensitivity.

Table 2 presents the characterization of chromosome 2 on the basis of the number of occurrences of *k-mers* for different k. We found that over 90% *k-mers* occur only once, however, a consistent number of *k-mers* can be encountered more than 1000 times. The maximum value ranged up to $1,644,958$ for few k-mers. Intuitively, it can be understood that to reduce the average number of candidate locations per read, *k-mers* that produce over 1000 occurrences should be extended. We assume a threshold of 1000 in CORAL, although, as this value is heuristic it can be changed. The impact of verification-aware *k-mer* length adaptation is discussed in Section 5.

Occurrence count	k-mer lengths						
	16	17	18	19	20	21	22
	Proportion of total k-mers in %						
One	91.68	95.13	96.47	97.02	97.29	97.46	97.59
≤100	8.30	4.85	3.51	2.96	2.69	2.52	2.40
≤1000	0.022	0.020	0.019	0.018	0.017	0.016	0.015
>1000	.0015	.0013	.0012	.0011	.001	.0009	.0008

TABLE 2
Charaterization of number of occurrences of *k-mers* for $k = 16, 17, 18, 19, 20, 21, 22$ in chromosome 2. Values given are in percentage approximated to the nearest decimal.

3.4 Kernel Algorithm: Search and Verification

Input to the kernel are: Integer encoded genome, reads, suffix array, tally matrix, modified F array and other constants. The constants include read length, minimum *k-mer* length, number of *k-mers*, δ and eb. Fig. 4 visualizes the algorithmic procedure followed by the kernel. It starts with loading read to the private memory followed by integer coding and storage of forward and reverse strand of the read. Integer encoding is performed to access the elements of a particular column and row of tally matrix and is performed using the following scheme: $\{(A:0),(C:1),(G:2),(T:3),(Z:4)\}$. After preprocessing of reads, we perform filtration and verification of forward strand followed by reverse strand of the read.

Filtration is divided into two stages: *Pre-search* and *Search*, as highlighted in Fig. 4. *Pre-search* is a preliminary search of all the *k-mers* using our proposed dynamic extension approach to identify the unused extra bases, u_eb, if any, and records the corresponding unextended *k-mer*. The unused bases remain when few or all *k-mers* are not extended as the number of candidate locations reported by them are within the threshold. These extra unused bases can still be used to extend some of the *k-mers* to further reduce the overall number of candidate locations reported by all the *k-mers*. For example, given $n = 100$, $\delta = 5$ and $eb = 4$, if each of 6 *k-mers* result in < 1000 candidate locations, then, $u\_eb = 4$. The goal, here, is to consume the entire read in filtration step irrespective of the *k-mer* lengths. Therefore, to utilize all the u_eb, we extend 4 out of 6 *k-mers* by one base each, increasing k to 17 from 16. The information on u_eb is used during the *Search* stage, each unextended *k-mer* is elongated by one base until $u\_eb = 0$. The choice of *k-mers* is serial starting from the first. This ensures that

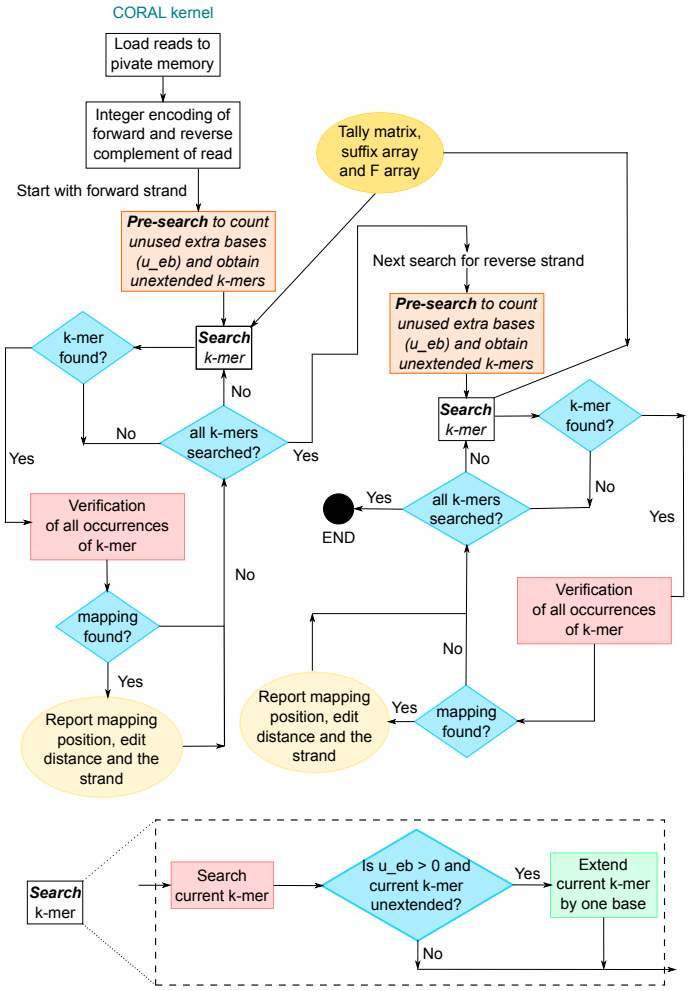

Fig. 4. Algorithm for the CORAL kernel.

all the extra bases are utilized to minimize the number of candidate locations. Both *Pre-search* and *Search* start from the end of the read and depending on the number of eb, u_eb and candidate locations, it extends the *k-mer* till the number of candidate locations are < 1000 or all the spare bases are exhausted. As *Search* proceeds, all the candidate locations of each *k-mer* are verified in-situ and the mappings found are reported. The same procedure is followed for the reverse strand.

4 Experimental Results

As stated in section 1, the host program of CORAL is written in `Python` and the kernel is in `C`. We use `PyOpenCL` rather than conventional `C`-based OpenCL because scripting languages, such as `Python`, enables fast modifications and prototyping and is more programmer friendly. We used OpenCL 1.2 standard to compile the kernel. We choose `Python` because it considerably eases string operations and manipulation, especially, the outlier operations which do not affect the assembly directly.

4.1 Experimental setup

We use both real and simulated reads to compare CORAL with RazerS3, Yara, Hobbes3 and FEM from the *all-mapper* category and BWA-MEM and GEM from the *best-mapper* category. We use a total of **6 million simulated reads** and **2 million real reads**. Wherever possible, only the mapping times and accuracy have been compared. We have mapped both simulated and real single-end reads to chromosome (chr) 2 and 21 of the human genome. The latest version of Mason [26] (`mason2-2.0.9`) is used to produce simulated reads. We use 12 sets of 500,000 reads each, 6 of them are derived from chr 2 and other 6 from chr 21. Out of the 6 sets, three of them have reads of length 100 and the other three have reads of length 150. The three sets, with read length of 100, are segregated based on edit distances with which they are sequenced from the chromosomes viz. 3 or less, 4 or less and 5 or less. Similarly, the reads of length 150 are segregated based on edit distance viz. 5 or less, 6 or less and 7 or less. Thus, resulting in a total of **6 million simulated reads** to be mapped by all the mappers. The chromosomes used in this paper are from the human genome version GRCh38/hg38, dated Dec. 2013, and were downloaded from the UCSC genome browser [27]. We used **1 million (M) real reads** each from NCBI ERR012100_1 and SRR826460_1 of length 100 and 150, respectively. We run all the mapping tools including CORAL on two separate systems with the following configurations:

System 1: Intel Core i5-6600 CPU @ 3.30GHz, 64GB RAM
System 2: Intel Core i7-2600 CPU @ 3.40GHz, 16GB RAM + 2 $\times$ GeForce GTX 590, 1.5 GB RAM

OpenCL computing framework imposes the following two restrictions:

a) OpenCL 1.2 standard does not permit dynamic memory allocation. Because of this, CORAL requires the number of outputs per read to be mentioned beforehand, in order to allocate sufficient memory for each read to store the mapping locations, strands and edit distances. Thus, it reports a maximum of *first-n* mapping locations per read as informed in Table 1.
b) It does not permit allocation of more than $(1/4)^{th}$ of the RAM capacity to a single variable. Example, with 16 GB RAM no variable can have more than 4GB of memory allocated. It limits both the size of the data structure to be stored and the number of outputs desired per read.

To elaborate further on (a), if a read matches only at few locations, it will still require to be allotted sufficient space for the given number of outputs desired per read. As we have limited RAM on system 2, especially, in the GPUs, the number of outputs per read must be assigned accordingly to ensure we do not run out of memory resource. As system 1 has large RAM capacity, we allot 3500 outputs per read to show the accuracy of CORAL and allot 100 outputs per read on system 2 to demonstrate speedups obtained by using multiple devices. The preprocessed tally matrix size depends on the length of the chromosome i.e. the size for chr 2 is 3.9GB which is much larger than that of 747.4 MB for chr 21. As GPUs have limited RAM size of 1.5 GB, the data structure for chr2 cannot be loaded on them. Hence, to demonstrate the implementation on multiple devices simultaneously, we use smaller chr21 to map the real and

simulated reads. In summary, we present the results for the following combinations:

CORAL on System 1: Both real and simulated reads are mapped to chr2 and chr21 with 3500 outputs per read for different number of errors, using only the CPU.

CORAL-cpu on System 2: Both real and simulated reads are mapped to chr2 and chr21 with 100 outputs per read for different number of errors, using only the CPU.

CORAL-all on System 2: Both real and simulated reads are mapped to chr21, only, with 100 outputs per read for different number of errors, using CPU along with the GPUs.

4.1.1 Estimating accuracy with respect to simulated reads

For the simulated reads, the SAM file obtained from Mason is used as the gold standard for measuring mapping accuracy. On system 1, CORAL, RazerS3, Hobbes3 and GEM report up to 3500 mapping locations per read while BWA-MEM, FEM and Yara report all the mapping locations, since they do not provide the facility to report fixed number of mappings. On system 2, all the mappers are configured to either report up to 100 mapping locations per read, wherever possible, or all the mapping locations. Simulated reads originate from a known position reported in the SAM file obtained from Mason. Hence, to determine the mapping accuracy, the output files from the mappers are parsed and searched for original mapping location, strand and edit distance. If any of outputs reported by the mappers match to that of the gold standard for a particular read, we record an accurate mapping. This procedure is followed for all the simulated reads and all the mappers under consideration. While comparison with the gold standard, we allow for a threshold, $\tau = \pm 10$ bases with respect to the original location. Irrespective of the chosen τ, the criteria for measuring a match remains uniform for all the mappers.

4.1.2 Estimating accuracy with respect to real reads

A similar approach is followed for real reads, however, the SAM file obtained from RazerS3 is used as the gold standard. We use RazerS3 as it has been used in Hobbes3, FEM and Yara to build the gold standard due to its high accuracy and *all-mapper* capability. On System 1, we use RazerS3 to produce SAM file with up to 1000 outputs per read and all the other mappers (including CORAL) are configured to report 3500 mapping locations per read, if possible, or all the mapping locations. Following that, we identify if all the, up to 1000, mapping locations per read reported in the gold standard are present in the output of other mappers. In comparison with section 4.1.1, where it is sufficient to find a single known location of origin of a simulated read, here, for the real reads all the locations reported (up to 1000) by the gold standard is compared with 3500 outputs of other mappers, thus, making the evaluation criteria relatively stringent.

On System 2, we do the opposite. We configure RazerS3 to produce up to 1000 outputs per read and configure the mappers to map up to 100 outputs per read. Here, we measure accuracy by identifying if all the reads mapped by the gold standard i.e. RazerS3, have been reported by other mappers with at least one matching mapping location, strand and edit distance. We limit the number of outputs to 100 because of the limitations on RAM capacity of System 2. Using the aforementioned configurations for evaluation of accuracy, we present results similar to the benchmarking method used in Rabema [28] i.e. the *all* and *all-best* scenario on System 1, and *any-best* scenario on System 2.

4.1.3 Configurations of read mappers

RazerS3: We used the latest version available viz. `razers3-3.5.8`. Pigeonhole filter was used with thread count of 16 for different percentage identity or error rates and number of outputs. The following provides an example of the command line parameters used:

```
razers -fl pigeonhole -tc 16 -i 95 -rr 100
-m 3500 -v -o OUTPUT.sam chr2.fa INPUT.fq
```

Yara: The latest version `1.0.2` was used with 16 thread in `full` sensitivity mode.

```
yara_mapper chr2.index INPUT.fq -v -e 4 -y
full -t 16 -o OUTPUT.sam
```

Hobbes3: We used latest version `3.0` with 16 threads and varying number of maximum number of outputs and errors per read.

```
hobbes -sref chr2.fa -i
chr2_hobbes3_index.hix -k 3500 --indel -q
INPUT.fq -v 5 -p 16 --mapout
OUTPUT.sam
```

FEM: We used latest FEM version available dated 03/13/2018. For index construction, we used window size of 12 and step size of 4 (e.g. `FEM index 12 4 chr2.fa`) and for mapping we used 16 threads with edit distance configuration. FEM, by default, reports all the mapping positions. The following provides an example of the command line parameters used:

```
FEM align -t 16 -f "vl" --ref chr2.fa --read
INPUT.fq -o OUTPUT.sam -e 5
```

GEM: We used the latest version `3`. For simulated reads, we run GEM in `sensitive` mapping mode, however, for real reads we used `fast` mapping mode, as `sensitive` took hours to produce results. We used 16 threads with varying number of outputs i.e. 100 or 3500, and error rates viz. 3 to 7. The following provides an example of the command line parameters used:

```
gem-mapper --index chr2.gem -v -t 16 -M 3500
--mapping-mode sensitive -i INPUT.fq
-o OUTPUT5.sam -e 0.05
```

BWA-MEM: We used latest BWA version `0.7.17`. We configured BWA-MEM to find all mapping locations with a thread count of 16. BWA-MEM is configured to skip *k-mers*(or seeds) with more than 500 occurrences by default. We increased it to 1000 similar to the threshold value used for CORAL. BWA-MEM does not permit specifying edit distance for read mapping unlike BWA-aln, hence, for real reads we could not obtain mapping times for different edit distance values. The following provides an example of the command line parameters used:

```
bwa mem -t 16 -c 1000 -v 3 -a chr2.fa
INPUT.fq > OUTPUT.sam
```

Chromosome 2	Read length	100						150					
	Error	3		4		5		5		6		7	
	Time/Accuracy	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)
System 1: Intel Core i5-6600 CPU@3.30GHz, 64GB RAM	RazerS3	169.2	99.99	215.8	99.99	265.9	100	190.7	99.99	224.5	100	294.3	99.99
	Hobber3	16.18	99.99	19.52	99.99	28.14	100	35.47	99.99	36.55	100	41.24	99.99
	FEM	3.30	41.18	4.58	41.10	6.51	41.17	5.71	37.32	7.49	37.47	9.67	37.48
	Yara	13.35	97.92	33.85	97.91	41.42	97.85	38.73	98.60	47.46	98.59	102.44	98.57
	BWA-MEM	53.30	99.71	61.53	99.32	66.96	98.65	74.35	99.61	80.46	99.36	85.39	99.03
	GEM	14	97.90	21	97.89	48	97.81	13	98.57	18	98.57	25	98.54
	CORAL-cpu	**11.89**	**99.77**	**19.35**	**99.71**	**30.99**	**99.75**	**23.24**	**99.91**	**34.50**	**99.91**	**45.04**	**99.92**
System 2: Intel Core i7-2600 CPU@3.40GHz, 16GB RAM + 2 × GeForce GTX 590, 1.5 GB	RazerS3	125.5	99.78	165.2	99.77	193.7	99.77	131.8	99.93	158.0	99.93	218.4	99.93
	Hobber3	11.70	99.77	10.58	99.76	10	99.73	32.54	99.90	28.80	99.88	24.86	99.86
	FEM	1.97	41.18	2.63	41.10	5.02	41.17	2.87	37.32	3.49	37.47	5.37	37.48
	Yara	8.87	97.92	25.04	97.91	28.92	97.85	28.34	98.60	33.07	98.59	72.1	98.57
	BWA-MEM	39.43	99.71	45.88	99.31	50.79	98.65	56.06	99.61	61.17	99.36	65.97	99.03
	GEM	7	97.90	12	97.89	27	97.81	7	98.57	11	98.57	14	98.54
	CORAL-cpu	**8.14**	**99.48**	**15.67**	**99.31**	**27.71**	**99.24**	**16.89**	**99.67**	**29.8**	**99.60**	**40.13**	**99.50**

TABLE 3
The results of mapping three sets of 500,000 simulated reads, with different maximum edit distances viz. 3,4 and 5, to chromosome (chr) 2 on the CPU. RazerS3, Hobbes3, GEM and CORAL-cpu reported up to 3500 mapping per read on System 1 and 100 outputs per read on System 2. FEM, Yara, however, by default report all the mapping positions and BWA-MEM was configured to report all mapping positions. T and A are abbreviations for time and accuracy and have been reported in seconds and proportional percentage of accurate mapping against the total number of reads.

Chromosome 21	Read length	100						150					
	Error	3		4		5		5		6		7	
	Time/Accuracy	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)
System 1: Intel Core i5-6600 CPU@3.30GHz, 64GB RAM	RazerS3	29.95	99.99	48.30	99.99	49.13	99.99	35.12	99.99	42.43	99.99	56.31	99.99
	Hobber3	13.42	99.99	14	99.99	17.19	99.99	31.75	99.99	30.73	99.99	32.13	99.99
	FEM	1.535	39.45	1.79	39.45	2.31	39.51	2.14	36.09	2.30	36.14	2.82	36.25
	Yara	10.58	92.71	18.57	92.64	22.62	92.58	27.75	94.25	32.07	94.17	41.09	94.13
	BWA-MEM	60.14	99.72	74.35	99.37	72.08	98.78	80.69	99.64	86.64	99.37	92.37	99.04
	GEM	10	92.69	13	92.59	20	92.54	11	94.12	15	94.10	17	94.04
	CORAL-cpu	**7.55**	**99.99**	**11.87**	**99.98**	**19.31**	**99.98**	**13.31**	**99.99**	**20.78**	**99.99**	**27.35**	**99.99**
System 2: Intel Core i7-2600 CPU@3.40GHz, 16GB RAM + 2 × GeForce GTX 590, 1.5 GB	RazerS3	23.39	99.74	31.42	99.72	39.21	99.70	26.52	99.98	33.39	99.98	46.53	99.98
	Hobber3	11.12	99.60	9.37	99.54	7.92	99.44	31.91	99.89	27.55	99.86	23.15	99.81
	FEM	1.16	39.45	1.29	39.45	1.59	39.51	1.59	36.09	1.73	36.14	1.89	36.25
	Yara	7.26	92.71	14.41	92.64	16.67	92.58	20.67	94.25	23.19	94.17	29.19	94.13
	BWA-MEM	43.27	99.71	51.16	99.37	57.55	98.78	73.98	99.64	78.05	99.37	85.05	99.04
	GEM	5	92.69	7	92.59	12	92.54	7	94.12	9	94.10	11	94.04
	CORAL-cpu	**4.42**	**99.43**	**8.12**	**99.25**	**14.85**	**99.10**	**9.13**	**99.70**	**15.01**	**99.53**	**21.88**	**99.30**
	CORAL-all	**3.09**	**99.43**	**5.40**	**99.25**	**8.55**	**99.10**	**6.15**	**99.70**	**9.20**	**99.53**	**13.04**	**99.30**

TABLE 4
The results of mapping three sets of 500,000 simulated reads, with different maximum edit distances viz. 3,4 and 5, to chromosome (chr) 21. RazerS3, Hobbes3, GEM and CORAL-cpu reported up to 3500 mapping per read on System 1 and 100 outputs per read on System 2. CORAL-all, also, produce 100 outputs per read but executes on CPU and both the GPUs, simultaneously. FEM, Yara, however, by default report all the mapping positions and BWA-MEM was configured to report all mapping positions. T and A are abbreviations for time and accuracy and have been reported in seconds and proportional percentage of accurate mapping against the total number of reads.

4.2 Results

4.2.1 Simulated reads mapped to chr 2

Table 3 presents the results of mapping three sets of 500,000 simulated reads to chr 2 on the CPU of System 1 and 2. The three sets of reads have different maximum error or edit distances viz. 3, 4 and 5, respectively. We can observe that CORAL is $5 - 16\times$ faster than RazerS3 and maps over 99% of reads, showing comparable accuracy. The runtime of CORAL is better than Hobbes3 for low error rates and comparable for higher error rates, for example, $n = 100, \delta = 5$ and $n = 150, \delta = 7$. On System 2, as the number of outputs is small, Hobbes3 performs better than CORAL for high error rates. The accuracy of both Hobbes3 and CORAL are comparable and are over 99%. CORAL outperforms Yara and BWA-MEM in all the cases, with up to $2.27\times$ and $4.84\times$ speed-up, respectively. CORAL beats GEM in runtime for $n = 100$ and accuracy, however, it lags for $n = 150$. FEM runtimes are faster than that of CORAL but it maps less than 40% of reads. GEM performs better as it is a *best-mapper* and designed to produce fewer accurate solutions. From the accuracy point of view, CORAL performs comparable to RazerS3, Hobbes3 and BWA-MEM and outperforms Yara, FEM and GEM in all cases. With regards to mapping time CORAL is better than Yara, BWA-MEM and RazerS3, and comparable to Hobbes3 and GEM. It, however, underperforms with respect to FEM, which is fast but the accuracy is low.

4.2.2 Simulated reads mapped to chr 21

Table 4 presents the results of mapping three sets of 500,000 simulated reads to chr 21 on the CPU of system 1 and 2

Chromosome 2	Read length	100						150					
	Error	3		4		5		5		6		7	
	Time/Accuracy	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)
System 1: Intel Core i5-6600 CPU@3.30GHz, 64GB RAM	RazerS3	229.7	100	334.4	100	443.5	100	268	100	388.6	100	631.6	100
	Hobber3	38.24	100	38.67	100	52.12	99.99	61.63	100	56.89	100	58.81	100
	FEM	4.80	0.65	6.87	0.44	12.14	0.32	5.23	0.20	6.73	0.83	9.33	0.11
	Yara	22.52	1.88	96.70	1.32	118.34	1.01	178	1.18	873.84	0.96	1971.8	0.83
	BWA-MEM	T(s) - 120.5			A(%) - 14.24			T(s) - 234.9			A(%) - 11.45		
	GEM	25	1.78	25	1.22	29	0.92	58	1.03	57	0.15	55	0.70
	CORAL-cpu	**31.19**	**96.05**	**52.75**	**95.77**	**89.09**	**94.04**	**68.57**	**99.89**	**125.6**	**99.5**	**187.88**	**97.72**
System 2: Intel Core i7-2600 CPU@3.40GHz, 16GB RAM + 2 × GeForce GTX 590, 1.5 GB	RazerS3	160.1	100	247.5	100	363	100	185.9	100	307.2	100	560.1	100
	Hobber3	24.92	100	24.37	100	27.85	100	63.49	100	56.36	100	52.60	100
	FEM	3.03	31.8	4.21	29.61	7.61	27.41	3.36	16.13	4.23	14.41	5.86	12.73
	Yara	13.44	100	74.17	100	87.15	99.99	136.5	100	824.9	100	1877	100
	BWA-MEM	T(s) - 85.44			A(%) - 97.49			T(s) - 161.6			A(%) - 93.51		
	GEM	22	94.54	22	92.97	21	91.33	55	86.80	52	86.93	52	84.34
	CORAL-cpu	**22.18**	**99.91**	**44.45**	**99.75**	**81.53**	**99.66**	**69.3**	**99.89**	**129.6**	**99.79**	**198.6**	**99.71**

TABLE 5
The results of mapping 1M real reads to chromosome (chr) 2 on the CPU. RazerS3, Hobbes3, GEM and CORAL-cpu reported up to 3500 mapping per read on System 1 and 100 outputs per read on System 2. FEM, Yara, however, by default report all the mapping positions and BWA-MEM was configured to report all mapping positions. T and A are abbreviations for time and accuracy and have been reported in seconds and proportional percentage of accurate mapping against the total number of reads.

Chromosome 21	Read length	100						150					
	Error	3		4		5		5		6		7	
	Time/Accuracy	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)
System 1: Intel Core i5-6600 CPU@3.30GHz, 64GB RAM	RazerS3	39.33	100	58.56	100	77.85	100	43.86	100	62.69	100	100.5	100
	Hobber3	24.57	100	24.1	100	22.22	100	53.31	100	45.75	100	38.31	100
	FEM	2.34	0.459	2.58	0.373	3.50	0.313	2.06	0.074	2.33	0.077	2.81	0.061
	Yara	12.58	100	26.65	100	36.18	100	52.48	100	135.9	100	357.58	100
	BWA-MEM	T(s) - 127.39			A(%) - 22.96			T(s) - 219.15			A(%) - 28.64		
	GEM	24	2.03	25	1.81	24	1.66	61	4.65	60	3.97	59	3.41
	CORAL-cpu	**11.68**	**99.93**	**21.39**	**99.87**	**42.14**	**99.91**	**21.52**	**100**	**45.33**	**100**	**75.80**	**100**
System 2: Intel Core i7-2600 CPU@3.40GHz, 16GB RAM + 2 × GeForce GTX 590, 1.5 GB	RazerS3	26.79	100	42.37	100	64.52	100	30	100	49.10	100	88.92	100
	Hobber3	20.31	100	16.87	100	14.44	100	58.36	100	49.88	100	40.7	100
	FEM	2.50	16.45	2.15	14.52	2.27	12.72	2.27	1.44	2.12	1.75	3.04	1.59
	Yara	6.09	100	18.59	100	24.34	100	35.77	100	110.7	100	309.7	100
	BWA-MEM	T(s) - 184.38			A(%) - 97.17			T(s) - 359.56			A(%) - 95.09		
	GEM	23	93.66	22	92.04	22	89.97	56	90.20	54	91.35	54	89.07
	CORAL-cpu	**7.72**	**100**	**18.33**	**99.98**	**39.79**	**99.99**	**20.18**	**100**	**45.38**	**100**	**81.42**	**100**
	CORAL-all	**5.36**	**100**	**12.41**	**99.98**	**27.95**	**99.99**	**12.5**	**100**	**27.41**	**100**	**49.6**	**100**

TABLE 6
The results of mapping 1M real reads to chromosome (chr) 21. RazerS3, Hobbes3, GEM and CORAL-cpu reported up to 3500 mapping per read on System 1 and 100 outputs per read on System 2. CORAL-all, also, produce 100 outputs per read but executes on CPU and both the GPUs, simultaneously. FEM, Yara, however, by default report all the mapping positions and BWA-MEM was configured to report all mapping positions. T and A are abbreviations for time and accuracy and have been reported in seconds and proportional percentage of accurate mapping against the total number of reads.

and CPU+GPU combination of system 2. As mentioned in previous section, the three sets of simulated reads have maximum error of 3, 4 and 5. Compared to RazerS3, CORAL is $2-8\times$ faster and maps over 99% of the reads accurately. CORAL outperforms Hobbes3 in all cases except for $n = 100$ and $e = 5$. It, considerably, outperforms FEM on mapping accuracy. From table 4, we can observe that CORAL-all, which distributes a portion of the workload on Nvidia GPUs, results in up to $2\times$ speedup with the same accuracy. CORAL outperforms Yara and BWA-MEM in all the cases on System 1, with up to $2.08\times$ and $7.96\times$ speed-up, respectively. For CORAL-all, we equally distributed 256,000 out of 500,000 reads on two Nvidia devices and the remaining 244,000 on the CPU to obtain speedups. From the experiments, we conclude that CORAL-cpu outperforms RazerS3, Hobbes3, GEM, Yara and BWA-MEM in most of the cases on either mapping time or accuracy or both and if not produce comparable results. Similar to section 4.2.1, FEM is faster than CORAL but lags considerably in accuracy. CORAL-all which uses multiple devices provide an additional speedup of up to $2\times$, unlike other mappers who are optimized to operate only on the CPU.

4.2.3 Real reads mapped to chr 2

System 1: Table 5 presents the results of mapping 1 M real reads on chr2, from two different databases with different read lengths. RazerS3, Hobbes3, GEM and CORAL-cpu reported up to 3500 mapping per read. We can observe that CORAL is $3-7\times$ faster than RazerS3 and accurately maps over 94% of reads. It is also evident that it is considerably better than FEM and GEM in accuracy of mapping reads. We could not run GEM in the sensitive mode for real reads as it was taking very long runs. For $\delta = 3$, CORAL outperforms Hobbes3, however, it lags behind for higher error rates. On the contrary, CORAL outperforms Yara on all cases, especially, for higher error rates, leaving single case where $\delta = 3$ and $n = 100$, and beats it on accuracy in all cases. CORAL beats BWA-MEM on all parameters and cases.

System 2: On System 2, all the mappers are configured to map up to 100 positions per read. Yet again it can be seen that CORAL outperforms Yara and BWA-MEM in all the cases on mapping times. On accuracy, leaving for a few cases with respect to Yara, with a marginal $< 0.4\%$ difference, CORAL outperforms both Yara and BWA-MEM. The accuracy for Yara, BWA-MEM, FEM and GEM are higher on System 2 due to different comparison criteria used, as discussed in Section 4.1.2. Here, we measure *any-best* accuracy of Rabema. For all cases, CORAL, considerably, outperforms GEM and FEM in accuracy.

4.2.4 Real reads mapped to chr 21

System 1: Table 6 presents the results of mapping 1 M real reads, from two different databases with different read lengths, on chr 21. On System 1, all mappers produce 3500 outputs per read, except, the RazerS3 which serves as the gold standard. We can observe that CORAL is $2-4\times$ faster than RazerS3 in all cases. Leaving for $n = 100, \delta = 5$, CORAL is up to $5\times$ faster than Yara with similar accuracy. It outperforms BWA-MEM on all accounts. CORAL mapping times are better than Hobbes3 and GEM for lower error rates except for $n = 100, \delta = 5$ and $n = 150, \delta = 7$. FEM and GEM reportedly mapped only a small number of reads, hence, CORAL mapping accuracy supersedes them.

System 2: On System 2, all mappers report up to 100 outputs per read, except, the RazerS3 which serves as the gold standard. The results, here, follow similar trend as explained above. However, we can see that mapping times can be halved if all the available resources are used with judicious workload distribution. CORAL-all executes on CPU and both the GPUs producing up to $2\times$ speedup. For $n = 150$, we mapped 368,000 reads on GPUs and remaining 632,000 reads on the CPU. For $n = 100$, we mapped 340,000 reads on the GPU and remaining 660,000 reads on the CPU. These numbers were chosen depending on the memory capacity of GPUs and kernel requirements. CORAL-cpu and CORAL-all outperform RazerS3 and BWA-MEM in all the cases. Except for $n = 100, \delta = 5$ and $n = 150, \delta = 7$, it outperforms Hobbes3, GEM and Yara in all other cases. In case of FEM, the mapping accuracy were found to be considerably low despite successive experiments.

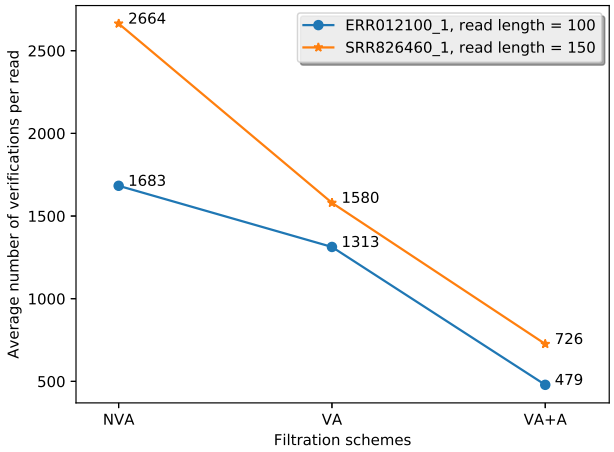

Fig. 5. Average number of verifications per read using different filtration schemes for real data sets, viz. ERR012100_1 ($n = 100, \delta = 5$) and SRR826460_1 ($n = 150, \delta = 7$), on chr2. NVA - non verification-aware, VA - verification-aware and VA+A - verification-aware along with approximation. The approximation used, here, limits the maximum number of verifications per *k-mer* to 1000.

4.2.5 Evaluation of Verification-Aware Filtration

Fig. 5 shows the average number of verifications performed per read using three filtration schemes: non verification-aware (NVA), verification-aware (VA) and verification-aware with approximations (VA+A). In NVA scheme, we fix the lengths of *k-mer* and calculate the total number of verifications required for all the reads. For example, given $n = 100, \delta = 5$ and $n = 150, \delta = 7$, the *k-mer* lengths are $(17, 17, 17, 17, 16, 16)$ and $(19, 19, 19, 19, 19, 19, 18, 18)$, respectively. In VA scheme, CORAL dynamically determines the lengths of *k-mer* by extending them depending on the number of verifications encountered. VA+A scheme is similar to VA with an additional condition that limits the maximum allowed candidate locations per *k-mer* to 1000. Experiments performed on both the real data sets show that the number of verifications reduces significantly (up to $3.67\times$) across the filtration schemes.

5 Discussion

From Section 4.2, we see that, even though GEM uses a similar approach of *k-mer* length variation on FM-Index, CORAL outperforms GEM on either mapping times, accuracy or both for different read lengths, errors and datasets. For real reads, GEM could not produce any output in the sensitive mode, even, after long runtimes. Compared to GEM, CORAL is an *all-mapper* with portability and flexibility to work on heterogeneous systems. We observe that Hobbes3 outperforms CORAL in few cases, like, with longer chromosome, chr2, and high error rates, $\delta = 5, 7$. One of the major reasons is that the latest mappers use Streaming SIMD Extensions (SSE) instruction set. SSE instruction set utilizes 128-bit registers to accelerate computations. It enables loading of multiple bit vectors into a machine word, therefore, accelerating the banded Myers bit-vector algorithm, which is a major bottleneck in read mappers. OpenCL abstracts different hardware manifestations of parallel architecture including SIMD, but does not support the SSE instructions,

yet, for portability on wide-spectrum of devices. Difference in performance between CORAL and Hobbes3 is, also, due to *k-mer* selection criteria. CORAL selects the maximum possible length for each *k-mer* with an objective to reduce the number of candidate location, of a particular *k-mer*, by increasing its length using excess bases, if available. While Hobbes3 uses a dynamic programming based filtration scheme.

As mentioned in Section 1, the state-of-the-art mappers have focused on algorithmic innovations and software optimizations targeting only the CPU. To use them on different hardware, such as GPU or FPGAs, will require to be either rewritten or tailored. K. Reinert et al [2] present a review of the existing methods and algorithms, and predict in their concluding remarks that further improvements in the assembling time will result from accelerators and coprocessors. S. Aluru and N. Jammula [9] present a review on hardware accelerators for genome assembly on FPGAs and GPUs. Darwin [29] is a FPGA based coprocessor for whole genome alignment aiming at aligning genomes of two or more species. The authors have reported significant improvements in performance/$ and sensitivity by employing ungapped seeds. Darwin differs from CORAL as it aligns two genomes while we are mapping reads to assemble genome. A similarity between the two is the use of approximate string search algorithms. GateKeeper [30] implements the filtration stage on the FPGA and reports speedups over existing filtration schemes. The FPGA implementation, however, suffers from flexibility in mapping parameters such as read length and permissible edit distance. FPGAs, also, lack in on-board memory and communication bandwidth for faster data transfer between host processors, RAM and the FPGA chip, thereby, lagging behind the CPU in performance unless a large or multiple FPGA chips are used. Additionally, any change in the parameters may require extensive recoding and verification cycles. Jeremie S. Kim et al [31] present processing-in-memory (PIM) approach towards acceleration of filtration stage by implementing their filter on a 3D-stacked DRAM. It aims to optimize the filtration algorithm for 3D-stacked memory with high memory bandwidth and PIM capabilities. This, however, limits its portability to other hardware architectures.

In the case of GPU acceleration, the kernels are, often, designed targeting specific GPU architecture, often, using vendor specified programming framework. GPU architecture is optimized for floating-point operations while genome assembly involves integer based operations; hence, GPU may or may not serve as the best possible choice for accelerating genome assembly. Thus, mappers optimized for just one platform, be it CPU, GPU or FPGA, are unable to use the advantage of all the available resources. GPUs, for example, accompany CPU in most of the modern platforms ranging from workstations to servers. To the best of our knowledge, CORAL, for the first time, demonstrates simultaneous usage of all available resources on a system without any additional programming effort and achieving up to $2\times$ speedups. We showcase this by simultaneously deploying kernels on a quad-core CPU and two Nvidia GPUs. CORAL determines the maximum number of workitems in a workgroup in multiples of 2, as recommended by the Khronos group for better performance. OpenCL, then, automatically determines the total number of workgroups.

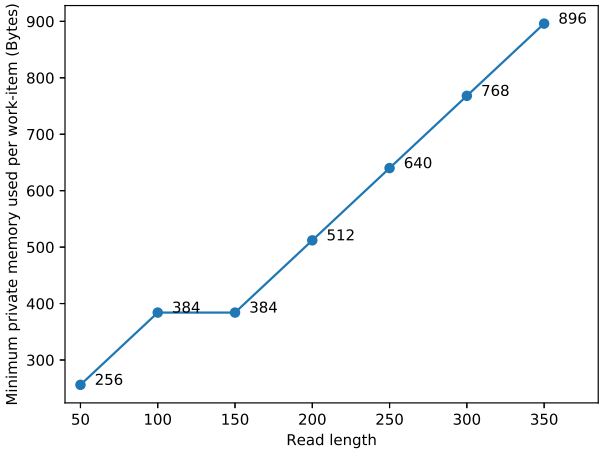

Fig. 6. The minimum amount of private memory, in bytes, used by each workitem in the kernel for different read lengths.

CORAL employs verification-aware filtration scheme which significantly reduces the average number of verifications performed per read, as shown in Fig. 5. The filtration methodology employing FM-Index backward search along with pigeonhole principle makes it fully sensitive, however, the approximation imposed in VA+A filtration scheme, i.e. limiting the maximum number of candidate locations per *k-mer* to 1000, as discussed in Section 4.1 and 4.2, permits for verification of maximum 12000 locations for both forward and backward strand combined. We observed that only about 3.22% and 4.77% of reads in ERR012100_1 and SRR826460_1, respectively, produce large numbers of candidate locations in the VA case, as can observed in Fig. 5. Although, the proportion of reads is small, however, number of the candidate locations produced per read is huge enough to skew the average number of verifications from 479 to 1313 and 726 to 1580, respectively. Therefore, we limit the maximum possible verification cycles to 1000.

CORAL, algorithmically, doesn't impose restrictions on the read lengths *per se*, however, in the current implementation it is practical to use it with short reads. This is because CORAL kernel loads read from the global memory to the private memory, as shown in Fig. 4, to reduce frequent memory accesses and from Fig. 6, we can see that the minimum private memory size used by each workitem in the kernel is proportional to the real length. In the current CORAL implementation, thus, the practicality of using longer reads depends on the availability of the private memory. CORAL does not produce the CIGAR string and SAM output format, yet. The memory footprint of CORAL is large due to tally and suffix array matrices, as mentioned in Section 4.1. These data structures, however, have the capability to significantly reduce their memory footprint as demonstrated in [21]. We envisage that our future work will have better *k-mer* selection methodology, SAM output and reduced memory footprint. With OpenCL based framework, CORAL can be run on credit-card sized single board computers (SBCs), designed for embedded scenarios. Such board have multicore architectures along with GPU, however, limited memory. All the compute units available in the form of CPU and

GPU can be simultaneously used using CORAL unlike other mappers proposed till date. We envisage aforementioned improvements as our future work.

6 Conclusion

In this paper, we presented, for the first time, a **C**ross-platf**O**rm **R**ead m**A**pper using openc**L** (CORAL) for heterogeneous systems. Such systems have different kinds of devices in various combinations on a single platform. For example, we present a case where a quad-core Intel CPU is accompanied by two Nvdia GTX 590 GPUs. To efficiently use all the compute resources on a system, CORAL employs OpenCL programming framework to launch kernels on the chosen devices and distributes the workload with the maximum possible workgroup size. Without any additional programming effort, CORAL can be used on any OpenCL conformant devices such as CPUs and GPUs, from different vendors, stiched together on embedded platforms, workstations and servers. Additionally, it uses a number of algorithmic optimisations, including verification-aware filtration, to significantly reduce the computational costs. We have used both simulated and real reads to compare the runtimes and accuracy of our tool with the state-of-the-art read mappers and showed that it offers competitive trade-offs besides portability. We envisage an improved filtration methodology, low memory footprint for data structures and SAM output capability as our future work.

Funding

The authors would like to thank School of Engineering (Newcastle Uni.) and Institute of Genetic Medicine (Newcastle Uni.) for studentship funding of the lead author and travel grant of the international collaborators. Further, all authors would like to acknowledge the support from EPSRC PRiME project (EP/K034448/1) and Royal Society Exchange project (IE161183).

Acknowledgments

The authors would like to thank Prof Saeid Nooshabadi (Michigan Tech., USA) and Prof S.K. Nandy (Indian Institute of Science, Bangalore, India) for the useful discussions feedback.

References

[1] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: Astronomical or genomical?" *PLOS Biology*, vol. 13, no. 7, pp. 1–11, 07 2015.

[2] K. Reinert, B. Langmead, D. Weese, and D. J. Evers, "Alignment of next-generation sequencing reads," *Annual Review of Genomics and Human Genetics*, vol. 16, no. 1, pp. 133–151, 2015, pMID: 25939052.

[3] L. Hood and D. Galas, "P4 medicine: Personalized, predictive, preventive, participatory a change of view that changes everything," *Computing community consortium*, 2008.

[4] S. Mittal and J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 69:1–69:35, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2788396

[5] S. Marco-Sola, M. Sammeth, R. Guigo, and P. Ribeca, "The gem mapper: fast, accurate and versatile alignment by filtration," *Nature Methods*, vol. 9, pp. 1185–1188, 2012.

[6] D. Weese, M. Holtgrewe, and K. Reinert, "Razers 3: Faster, fully sensitive read mapping," *Bioinformatics*, vol. 28, no. 20, pp. 2592–2599, 2012.

[7] J. Kim, C. Li, and X. Xie, "Hobbes3: Dynamic generation of variable-length signatures for efficient approximate subsequence mappings," pp. 169–180, 2016.

[8] H. Zhang, Y. Chan, K. Fan, B. Schmidt, and W. Liu, "Fast and efficient short read mapping based on a succinct hash index," *BMC bioinformatics*, vol. 19, no. 1, p. 92, 2018.

[9] S. Aluru and N. Jammula, "A review of hardware acceleration for computational genomics," *IEEE Design Test*, vol. 31, no. 1, pp. 19–30, Feb 2014.

[10] G. Khronos. (2018) List of opencl conformant products. https://www.khronos.org/conformance/adopters/conformant-products/opencl. Accessed: 2018-08-12.

[11] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*, 2000, pp. 390–398.

[12] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993. [Online]. Available: http://dx.doi.org/10.1137/0222058

[13] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *J. ACM*, vol. 46, no. 3, pp. 395–415, May 1999.

[14] H. Hyyrö, "A bit-vector algorithm for computing levenshtein and damerau edit distances," *Nordic J. of Computing*, vol. 10, no. 1, pp. 29–39, Mar. 2003.

[15] A. Prakash, S. Wang, A. E. Irimiea, and T. Mitra, "Energy-efficient execution of data-parallel applications on heterogeneous mobile platforms," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*, Oct 2015, pp. 208–215.

[16] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–10.

[17] M. Damschen, F. Mueller, and J. Henkel, "Co-scheduling on fused cpu-gpu architectures with shared last level caches," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2337–2347, Nov 2018.

[18] A. K. Singh, A. Prakash, K. R. Basireddy, G. V. Merrett, and B. M. Al-Hashimi, "Energy-efficient run-time mapping and thread partitioning of concurrent opencl applications on cpu-gpu mpsocs," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 147:1–147:22, Sep. 2017. [Online]. Available: http://doi.acm.org/10.1145/3126548

[19] E. Siragusa, "Approximate string matching for high-throughput sequencing," *Free University of Berlin*, 2015.

[20] H. Li and R. Durbin, "Fast and accurate long-read alignment with burrowswheeler transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, 2010. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btp698

[21] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature methods*, vol. 9, no. 4, pp. 357–359, 2012.

[22] H. Xin, S. Nahar, R. Zhu, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu, "Optimal seed solver: optimizing seed selection in read mapping," *Bioinformatics*, vol. 32, no. 11, pp. 1632–1642, 2016.

[23] M. Scarpino, *OpenCL in Action: How to accelerate graphics and computations*. Manning Publications, 2011.

[24] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equipment Corporation, Palo Alto, CA, Tech. Rep. 124, 1994.

[25] B. Langmead. (2014) Teaching materials: video lectures. ONLINE: http://www.langmead-lab.org/teaching-materials/.

[26] M. Holtgrewe, "Mason–a read simulator for second generation sequencing data," *Technical Report FU Berlin*, 2010.

[27] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, Haussler, and David, "The human genome browser at ucsc," *Genome Research*, vol. 12, no. 6, pp. 996–1006, 2002.

[28] M. Holtgrewe, A.-K. Emde, D. Weese, and K. Reinert, "A novel and well-defined benchmarking method for second generation read mapping," *BMC Bioinformatics*, vol. 12, no. 1, p. 210, May 2011.

[29] Y. Turakhia, S. D. Goenka, G. Bejerano, and W. J. Dally, "Darwin-wga: A co-processor provides increased sensitivity in whole genome alignments with high speedup," pp. 359–372, Feb 2019.

[30] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "GateKeeper: a new hardware architecture for accelerating pre-alignment in DNA short read mapping," *Bioinformatics*, vol. 33, no. 21, pp. 3355–3363, 05 2017. [Online]. Available: https://doi.org/10.1093/bioinformatics/btx342

[31] J. S. Kim, D. Senol Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "Grim-filter: Fast seed location filtering in dna read mapping using processing-in-memory technologies," *BMC Genomics*, vol. 19, no. 2, p. 89, May 2018. [Online]. Available: https://doi.org/10.1186/s12864-018-4460-0

Sidharth Maheshwari Sidharth Maheshwari received the Bachelor of Technology degree in Electronics and Electrical Engineering from Indian Institute of Technology (IIT) Guwahati, India in 2013. He has conducted research at IIT Hyderabad and Nanyang Technological University, Singapore between 2014 to 2016. Currently, he is working towards the PhD degree at the School of Engineering, Newcastle University, UK. He is currently working energy-efficient and performance-driven implementation of computational pipelines of whole genome sequencing on embedded platforms. His research interests include VLSI architectures, Bioinformatics and Biomedical Engineering. His published works can be found on his google scholar profile, link: https://scholar.google.co.uk/citations?hl=en&user=iKvALkYAAAAJ.

Venkateshwarlu Y. Gudur Venkateshwarlu Y. Gudur received the BE degree in Electronics and Telecommunication from Walchand Institute of Technology, Solapur and the M.Tech degree in VLSI Design from Shri Ramdeobaba College of Engineering and Management, Nagpur, in 2012 and 2014 respectively. Currently, he is working towards the PhD degree in Microelectronics and VLSI at the Department of Electrical Engineering, Indian Institute of Technology (IIT) Hyderabad, India. His research interests include hardware acceleration in healthcare applications, VLSI architectures, multiprocessor SoC and reconfigurable computing.

Rishad Shafik Rishad Shafik (MIET, MIEEE) is a Lecturer in Electronic Systems within the School of Engineering, Newcastle University, UK. Dr. Shafik received his Ph.D., and M.Sc. (with distinction) degrees from Southampton in 2010, and 2005; and B.Sc. (with distinction) from the IUT, Bangladesh in 2001. He is one of the editors of the book "Energy-efficient Fault-tolerant Systems," published by Springer USA. He is also author/co-author of 85+ IEEE/ACM journal and conference articles, with three best paper nominations. He has recently co-chaired 30th DFT2017 (www.dfts.org) at Cambridge, UK, and is a TPC member of DATE (www.date-conference.com) 2019, Florence, Italy. His research interests include energy-efficiency and adaptability aspects of embedded computing systems.

Ian Wilson Ian Wilson (CStat) is a lecturer in Statistical Genetics within the Institute of Genetic Medicine at Newcastle University. He has an MSc (distinction) in Statistics friom Sheffield University and a PhD in Population Genetics from Edinburgh University. His work has been in the application of computer intensive statistical methods to problems in the biological and medical sciences. His current research interests are at interface between Statistics and Medical Genomics. Dr Wilson is the author/co-author of 60 journal articles across Statistics, Genetics and the Biomedical Sciences.

Alex Yakovlev Alex Yakovlev (FIET, FIEEE, FRAEng) is a Professor of Computer Engineering, who founded and leads the MicroSystems Research Group, and co-founded the Asynchronous Systems Laboratory at Newcastle University. He was awarded an EPSRC Dream Fellowship in 2011–13. He has published 8 edited and co-authored monographs and more than 300 papers in IEEE/ACM journals and conferences, in the area of concurrent and asynchronous systems, with several best paper awards and nominations. He has chaired organizational committees of major international conferences. He has been principal investigator on more than 30 research grants and supervised 40 PhD students. Most recently, he has been elected to the fellowship of Royal Academy of Engineering in the UK.

Amit Acharyya Amit Acharyya received the Ph.D. degree from the School of Electronics and Computer Science, University of Southampton, U.K., in 2011. He is currently an Associate Professor with IIT Hyderabad, Hyderabad, India. His research interests include signal processing algorithms, VLSI architectures, low power design techniques, computer arithmetic, numerical analysis, linear algebra, bio-informatics, and electronic aspects of pervasive computing.