Arioc User Guide

written by
Richard Wilton

richard.wilton@jhu.edu

in collaboration with
Tamás Budavári
Ben Langmead
Sarah Wheelan
Steven L. Salzberg
Alexander S. Szalay

Copyright and other notices

Arioc User Guide

Introduction

Arioc is a set of computer programs that carry out the alignment of short DNA sequences to a comparatively large reference sequence or genome.

In the field of biological sequence alignment, there have been numerous attempts to accelerate successful CPU-based algorithms and implementations by using GPU (general-purpose graphics processing unit) hardware. A decade of such efforts has demonstrated that this is not easy. Arioc is interesting because it was engineered from the ground up to use GPU hardware to maximize the speed of biological sequence alignment without sacrificing accuracy.

It's hard to build a fast GPU-based read aligner

GPU programming requires software-development techniques that aren't needed for programs that run exclusively on a CPU. GPU hardware can run tens of thousands of threads of execution concurrently, but speed improvements in GPU software are not obtained simply by replicating source code that runs efficiently on one or two dozen CPU threads.

There are two fundamental reasons for this. One has to do with the way in which GPUs manage threads. In NVidia's CUDA environment (for which Arioc is implemented), threads execute in fixed groups of 32, in which each thread simultaneously executes the same instruction on different data. This SIMD (single instruction, multiple data) model impacts the layout of data in memory as well as the structure of source code. Algorithms that operate efficiently in CPU threads are often suboptimal when executed in SIMD threads on a GPU. In particular, algorithms that contain a significant amount of branching logic must be carefully redesigned so as to accommodate SIMD parallelism.

But the second, and probably more important, reason why GPU application speeds don't simply correlate with the number of available GPU threads is that there isn't enough memory bandwidth to keep up with all of those threads. GPU programming involves writing code that conforms to a variety of memory-layout and memory-addressing constraints that can fundamentally change assumptions that are taken for granted in algorithms that run efficiently on CPUs. Of course, memory optimization techniques are important in CPU software engineering as well, but the idiosyncrasies of GPU memory management tend to dominate other considerations in building a successful GPU application.

This all means that, in practice, GPU programmers are satisfied when they see a 10-fold speedup in a GPU-based application compared to a multi-threaded CPU-based application that performs the same computational task. (A hundred-fold GPU-versus-CPU speedup would be extraordinary and make one wonder whether the corresponding CPU application was properly optimized!) As far as Arioc is concerned, we did not release the software until we reached this 10x threshold.

What's different about Arioc

The central problem in read alignment is basically that of quantifying the similarity between two strings of symbols. This problem has attracted a great deal of attention since the early 1980s, and a number of successful algorithms have been developed in response. In particular, two algorithmic approaches —

one based on dynamic programming [Smith and Waterman, 1981] and the other using Levenshtein edit distance [Ukkonen, 1983] — are by far the most commonly used algorithms in modern read aligners. Arioc uses the Smith-Waterman algorithm, which consistently finds better mappings for reads that contain multiple insertions and deletions [Wilton et al, 2015], even though it requires more computational effort than edit-distance alignment.

Unfortunately, these string similarity computations are time-consuming and use a large amount of memory. With a reference sequence the size of the human genome, it is far from practical to align a read by computing similarity at every possible reference-sequence location. For this reason, read aligners perform a great deal of preliminary work so as to narrow down the number of reference-sequence locations at which they must compute alignments. This work cannot easily be reduced to a single algorithmic description. Instead, read aligners rely on heuristics and on software-engineering considerations to limit the number of reference-sequence locations they examine (and ultimately the number of string-similarity computations they perform) for each read.

It is here that we can recognize an opportunity to apply GPU-based parallelism to improve performance. Arioc implements a well-known technique known as "seed and extend" [Altschul, 1992] in a way that the basic operations of table lookup and prioritization of potential mapping locations are represented as sorting and adjacent-neighbor comparisons on long, one-dimensional arrays of integers. Such operations are well suited to SIMD implementation on GPU hardware. Arioc's speed and accuracy stem from GPU acceleration of this aspect of the read-alignment process.

What Arioc is (and isn't)

Arioc is a read aligner that uses GPU acceleration in an interesting way, namely, to find high-priority locations at which to compute alignments. The process of identifying high-priority alignment locations can be a significant bottleneck in a read aligner. Arioc uses GPU hardware to speed up this process and thereby achieve higher throughput [Wilton et al, 2015].

What Arioc does

Arioc accepts as input a large number — typically, hundreds of millions — of short sequencing reads. For each such read, Arioc reports the location (or locations) within a given reference sequence where a highly-similar subsequence is found.

Ideally, Arioc would find a perfect and unique mapping for every read. In other words, there would be exactly one location in the reference sequence where the read sequence exactly matches the reference sequence starting at that location. In practice, of course, many reads do not map perfectly or uniquely within a given reference sequence. In this case, Arioc reports what it considers to be the "best" mapping (or mappings) for each read. Arioc does these things accurately and with high throughput.

Arioc has several "database-oriented" features in its implementation. One of Arioc's original design goals was to carry out read alignment on sequence data residing in relational database tables. For this reason, Arioc handles input and output files that are encoded so as to facilitate data transfers to and from a database management system. (Arioc's encoded input and output formats are compatible with the binary data formats used by the Microsoft SQL Server for high-speed "bulk" input and output.) This may seem like "wasted time" if the goal is simply to align sequencer reads without archiving them in a database, but in fact the additional effort involved in encoding has little effect on throughput because Arioc uses CPU threads for file I/O concurrently with the rest of the GPU-based read-processing pipeline.

Arioc was designed to handle large amounts of sequencer data and genomes the size of the human reference genome. We remembered the notion of "scarcity thinking" [Cooper, 2004] and made a conscious effort to avoid it! If you are processing sequencer reads on a terabyte or petabyte scale, we assume that you have computers with the compute and memory resources required by Arioc.

What Arioc does not do

Arioc is not a tool for searching a database of short sequences and ranking those sequences by their similarity to a given sequence. (CUDASW++ [Liu et al, 2013] is the best-known GPU-accelerated tool for this.) Arioc does not operate with any sequences other than DNA; it does not understand any sequence "alphabet" other than ACGTN. It does not report SNPs or attempt to recognize variations. It does not do multiple sequence alignment or assemble sequencing reads in any way.

Finally, Arioc is not by any means a "simple", "turnkey", or "one size fits all" tool. Instead, Arioc exposes a variety of user-configurable parameters that let you experiment to optimize speed and sensitivity (or trade one for the other). The downside to this approach is that it exposes the complexity of the read-

alignment process; the benefit lies in the ability to fine-tune Arioc's performance to your own requirements.

The GPU software-development landscape is still changing. We expect Arioc to grow and adapt to newer, faster GPU hardware and software as it becomes available. We think that Arioc's approach to the read-alignment problem is sound, and that it has the potential to prove its utility as sequencers generate ever-increasing numbers of reads to align.

Installation

Arioc is written in C++ and compiled for the NVidia CUDA environment. The same code base is used for both Windows and Linux.

Hardware requirements

Arioc requires the following minimum hardware:

- a multi-core 64-bit Intel-compatible CPU
- at least 72 gigabytes of free memory
- sufficient disk space to store reference sequences, reads, and alignment results
- at least one NVidia GPU of compute capability 3.5 or greater, with at least 5 gigabytes of GPU memory

We have tested Arioc on both GTX-series and Kepler-series GPUs.

CUDA

Arioc has been tested with the following minimum CUDA versions:

- NVidia video driver v354.56
- CUDA runtime version 7.5

Operating systems

Arioc requires a 64-bit operating system. It has been tested with the following:

- Windows 7 (64-bit)
- Windows Server 2008 R2 datacenter
- Windows Server 2012 R2 datacenter
- RHEL Scientific Linux v7.2

Installation

Windows

Install using the Windows installer and the Arioc setup package (AriocSetup.msi). For source code only, unzip the Arioc.w.110.zip archive.

Linux

Unzip the Arioc.x.110.zip archive and build from source code.

Building from source code

The source code for is available in an archive whose name embodies the operating system and release version number. The sources for Windows release 1.10 are found in `Arioc.w.110.zip`; the Linux sources for the same release are in `Arioc.x.110.zip`.) The same sources can be compiled for Windows and for Linux:

- In Windows, use Microsoft Visual Studio 2013 or later to load the Arioc solution (`Arioc.sln`). You may need to edit the associated `.vsproj` files to reference the version of the CUDA SDK installed in your development environment.
- In Linux, use GNU make and the makefile in the top-level directory of the source-code distribution. Build each Arioc binary separately:

```
make clean
make AriocE
make AriocU
make AriocP
```

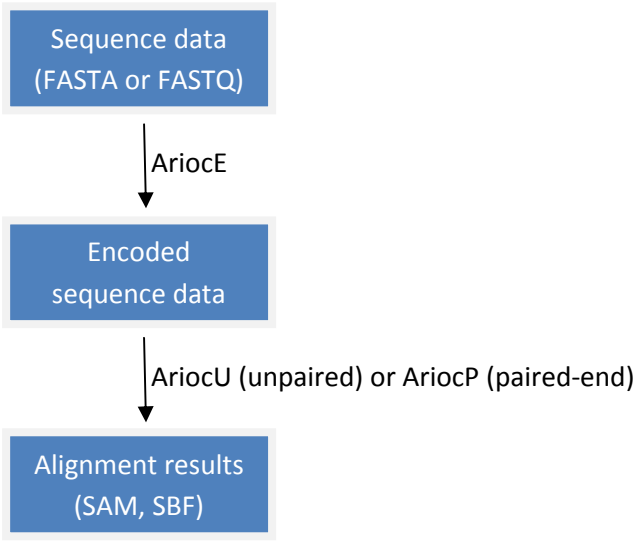
- You may verify your Arioc installation using the test data distributed in the `Arioc.RQA.zip` archive. The `readme.txt` file in this archive contains additional details.

Arioc runtime roadmap

There are three executable binaries in the Arioc distribution:

- AriocE: encodes sequence data
- AriocU: aligns unpaired reads
- AriocP: aligns paired-end reads

Here is how they are used:



The three executables are described in detail in the sections that follow.

Arioc command-line syntax

Each of the Arioc executables obtains its runtime configuration from an XML-formatted configuration file, so the command-line syntax for running them is simple:

```
AriocE config_filename
AriocU config_filename
AriocP config_filename
```

where `config_filename` is the name of the configuration file to use. (You should, of course, use fully-qualified filenames whenever it makes sense to do so.)

Arioc error messages

In general, Arioc error messages contain condensed information that identifies the source-code location that detected the error, along with human-readable text that explains the nature of the error. For example, if Arioc cannot open the configuration file you specify, it emits an error message like the following:

```
44:37.416 [00001704] ApplicationException ([0x5892] AriocAppMainBase.cpp 715):
XMLDocument::Loadfile( "AriocP.cfg" ) failed with error 3: file not found
XML error details:
AriocP.cfg
```

Arioc input and output

Arioc writes status and error messages to the `stderr` device. In practice, this means that all output appears in the console by default. If you want to redirect console output to a file, use command-line redirection syntax. For example:

```
AriocU test23.cfg 2> test23.log
```

For sequence input and alignment results, Arioc uses files whose names you specify in the configuration file.

AriocE: encoding sequence data

The AriocE component transforms raw sequence data from FASTA or FASTQ format into the binary format read by the AriocU and AriocP executables. Encoded sequence data resides in separate files whose contents can be distinguished by filename extension:

- `$a21.sbf, $a21.rc.sbf`: encoded forward and reverse-complement bases (3 bits per base, 20 bases per 64-bit value)
- `$raw.sbf, $raw.rc.sbf`: ASCII representation of the sequence data (forward and reverse complement)
- `$sqm.sbf, $sqm.rc.sbf`: sequence metadata (FASTA or FASTQ description line)
- `$sqq.sbf, $sqq.rc.sbf`: sequence quality scores (for FASTQ-formatted input only)

For example, encoded sequence data for human chromosome 1 might reside in the following set of files:

```
hs_ref_GRCh38_chr1$a21.sbf
hs_ref_GRCh38_chr1$a21.rc.sbf
hs_ref_GRCh38_chr1$raw.sbf
hs_ref_GRCh38_chr1$raw.rc.sbf
hs_ref_GRCh38_chr1$sqm.sbf
hs_ref_GRCh38_chr1$sqm.rc.sbf
```

Before using the Arioc aligners, AriocU and AriocP, you must encode both a reference sequence and a set of reads to align. Encoding is a one-time-only procedure. You can re-use the encoded files for multiple invocations of either the AriocU or AriocP executables.

Encoding unpaired reads

Encoding reads with AriocE is straightforward. For each FASTA file, AriocE generates an $a21, $raw, and $sqm file; for FASTQ input, AriocE generates a $sqq file as well.

The configuration file you provide to AriocE, in addition to specifying input and output filenames, contains "metadata" that is passed to the AriocU or AriocP aligners:

```
<?xml version="1.0" encoding="utf-8"?>

<AriocE>

  <dataIn sequenceType="Q">
    <file subId="1">C:\test103\i100p_1.fastq</file>
  </dataIn>

  <dataOut>
    <path>C:\BulkData\Q\Mason\test103</path>
  </dataOut>

</AriocE>
```

Simple configuration file for AriocE (unpaired reads).

In the XML configuration file, the `sequenceType=` attribute of the `<dataIn>` element indicates that the input sequences are to be encoded as reads. The <dataIn> element also contains one or more `<file>` elements, each of which specifies the fully-qualified name of an input file. Similarly, the <dataOut> element contains a `<path>` element that contains the fully-qualified name of the output directory.

AriocE requires that you use the `subId=` attribute of each `<file>` element to assign a "subunit ID" number to each input file. (Think of the subunit ID as a chromosome number for a reference genome or a sample subset identifier for a set of reads.) The subunit ID can be any integer between 1 and 127. AriocE combines the subunit ID, the data-source ID (specified as `srcId=` in the `<dataIn>` element), and an ordinal to create a unique 64-bit integer sequence identifier for every input sequence in every file you specify.

Although it was originally intended to simplify the automation of the sequence-alignment process, the use of an XML configuration file provides flexibility in parameterization as well as an opportunity to include human-readable documentation alongside runtime parameters. Here is an example of a more complex configuration file:

```
<?xml version="1.0" encoding="utf-8"?>

<AriocE maxDOP="2">

  <!-- Note: test 103 data consists of 100nt reads simulated using Mason -->
  <dataIn sequenceType="Q" srcId="1" samplingRatio="0.1">
    <rg CN="JHU PHA" DS="test 103" PL="ILLUMINA" />
    <file subId="1" ID="1" SM="sample 1: 11221">C:\test103\i100p_1.fastq</file>
    <file subId="2" ID="2" SM="sample 1: 11222">C:\test103\i100p_2.fastq</file>
  </dataIn>

  <dataOut>
    <path>C:\BulkData\Q\Mason\test103</path>
  </dataOut>

</AriocE>
```

Configuration file for AriocE (unpaired reads) with comments, optional parameters (XML attributes), and SAM-style read groups.

This configuration encodes two different files. It also specifies an optional parameter (`maxDOP=`) that controls the number of concurrent CPU threads used by AriocE; with a multi-core CPU and multiple input files, AriocE can encode multiple files concurrently and thereby run faster than it would on a single CPU thread. Finally, the `samplingRatio=` parameter causes AriocE to encode a randomly-sampled subset of the input reads:

This configuration file also illustrates how AriocE associates read group information with reads. AriocE merges the specified read group attributes (CN, DS, PL, ID, and SM) and copies them to its own "private" metadata file, which resides in the output directory. The AriocU and AriocP aligners use the read group information to generate appropriate @RG headers and RG tags in SAM-formatted output.

Here is a summary of the optional XML elements and attributes that control the runtime configuration of AriocE:

Element	Attribute	Description
<AriocE>	maxDOP	Number of concurrent CPU threads
<dataIn>	srcId	Data source ID (between 0 and 4,194,303)
	samplingRatio	Sampled fraction of reads encoded (maximum 1.0)
	QNAME	Encodes only the parenthesized fields in the FASTQ defline
<rg>	ID CN DS DT FO KS LB PG PI PL PU SM	SAM read group fields (common to all input files)
<file>	ID CN DS DT FO KS LB PG PI PL PU SM	SAM read group fields (merged with fields in the <rg> element)

Optional elements and attributes in the AriocE configuration file for encoding unpaired reads.

Both the QNAME= attribute in the <dataIn> element and the ID= attribute in the <rg> element accept a parenthesized wildcard specification that parses subfields from the FASTQ defline associated with each read. This makes it possible to generate meaningful SAM-formatted QNAME and RG (read-group) identifiers using the information in the FASTQ defline for each read. For example, given a read with a CASAVA-formatted FASTQ defline like this [Illumina, 2015]:

```
ERR037901.323683141 509.8.45.13203.181161/1
```

you can specify <dataIn ... QNAME="(*) *(/*)"> to cause AriocE to generate the QNAME string ERR037901.323683141/1. Similarly, you can use <rg ... ID="* (*.*)."> to associate the read with a read group whose ID is 509.8.

Encoding paired-end reads

The AriocE configuration file for paired-end reads uses the same parameters as the configuration file for unpaired reads, but it uses an additional XML attribute (mate=) to indicate the files in which mate 1 and mate 2 are found.

```
<?xml version="1.0" encoding="utf-8"?>

<AriocE>

  <dataIn sequenceType="Q">
    <file subId="1" mate="1">C:\test103\paired\i100p_1.fastq</file>
    <file subId="1" mate="2">C:\test103\paired\i100p_2.fastq</file>
  </dataIn>

  <dataOut>
    <path> C:\BulkData\Q\Mason\test103</path>
  </dataOut>

</AriocE>
```

Simple configuration file for AriocE (paired-end reads).

You can encode multiple input files (or, more exactly, pairs of input files) by using the subId= and mate= attributes of the <file> element:

```
<?xml version="1.0" encoding="utf-8"?>

<AriocE maxDOP="6">

  <dataIn sequenceType="Q" srcId="110114" samplingRatio="0.01">
```

```
    <file subId="5" mate="1">E:\yh110114\s_5_1_sequence.txt</file>
    <file subId="5" mate="2">E:\yh110114\s_5_2_sequence.txt</file>
    <file subId="6" mate="1">E:\yh110114\s_6_1_sequence.txt</file>
    <file subId="6" mate="2">E:\yh110114\s_6_2_sequence.txt</file>
    <file subId="7" mate="1">E:\yh110114\s_7_1_sequence.txt</file>
    <file subId="7" mate="2">E:\yh110114\s_7_2_sequence.txt</file>
  </dataIn>

  <dataOut>
    <path>C:\BulkData\Q\yh110114</path>
  </dataOut>

</AriocE>
```

Configuration file for AriocE (paired-end reads) with multiple input files and optional parameters.

Encoding a reference sequence

Encoding a reference sequence or genome is similar to encoding reads in that AriocE creates binary files that encode the sequence data and metadata. For a reference sequence or genome, however, AriocE also builds a set of hash tables that are used in both AriocU and AriocP.

Input files

If you are encoding a genome that consists of two or more reference sequences (chromosomes or other genome subunits), place each reference sequence in a separate file prior to encoding. For example, to encode the human genome, use a separate input file for each chromosome rather than concatenating all of the chromosome sequences into a single file.

AriocE recognizes reference sequence files in FASTA format only.

Two-pass encoding

AriocE builds hash tables in a straightforward manner: At every location in each reference, the encoder extracts a seed whose length and pattern is specified in the configuration file. A numerical hash function is applied to each seed and the reference-sequence location is appended to a list of such locations for the seed's hash value. Other configuration-file parameters control the number of bits in a hash value (and thus the size of the hash table) as well as the maximum number of reference-sequence locations recorded for a given hash value.

Internally, both the AriocU (unpaired) and AriocP (paired-end) aligners implement a pipeline in which nongapped alignments for each read are computed prior to gapped alignments. The aligners use different hash tables for nongapped (spaced-seed) alignment and for gapped (seed-and-extend) alignment. This means that you must execute AriocE twice to fully encode a reference sequence. The idea is to make it possible to use AriocE multiple times to create hash tables with different characteristics and then choose among those hash tables when you execute the AriocU or AriocP aligners.

Output directory layout

The output from each invocation of AriocE should occupy a single directory that contains encoded sequence files as well as a subdirectory for each hash table. If, for example, you run AriocE twice (once

for a nongapped hash table and once for a gapped hash table), and configure it to write its output to a directory named /GRCh38, the resulting directory layout would look like this:

```
/GRCh38              encoded sequence files (*.sbf)
/GRCh38/ssi84_2_30   hash table for nongapped (spaced-seed) aligner
/GRCh38/hsi20_0_30   hash table for gapped (seed-and-extend) aligner
```

Here is an example of an AriocE configuration file that builds a nongapped (spaced-seed) hash table for the human genome:

```
<?xml version="1.0" encoding="utf-8"?>

<AriocE seed="ssi84_2_30" maxDOP="32" maxJ="*">

  <dataIn sequenceType="R" srcId="0" filePath="/dss004/R/GRCh38.p6" uriPath="
ftp://ftp.ncbi.nlm.nih.gov/genomes/all/GCA_000001405.21_GRCh38.p6/GCA_000001405.21_GRCh38.p6_a
ssembly_structure/Primary_Assembly/assembled_chromosomes/FASTA">
    <file subId="1">chr1.fna</file>
    <file subId="2">chr2.fna</file>
    <file subId="3">chr3.fna</file>
    <file subId="4">chr4.fna</file>
    <file subId="5">chr5.fna</file>
    <file subId="6">chr6.fna</file>
    <file subId="7">chr7.fna</file>
    <file subId="8">chr8.fna</file>
    <file subId="9">chr9.fna</file>
    <file subId="10">chr10.fna</file>
    <file subId="11">chr11.fna</file>
    <file subId="12">chr12.fna</file>
    <file subId="13">chr13.fna</file>
    <file subId="14">chr14.fna</file>
    <file subId="15">chr15.fna</file>
    <file subId="16">chr16.fna</file>
    <file subId="17">chr17.fna</file>
    <file subId="18">chr18.fna</file>
    <file subId="19">chr19.fna</file>
    <file subId="20">chr20.fna</file>
    <file subId="21">chr21.fna</file>
    <file subId="22">chr22.fna</file>
    <file subId="23">chrX.fna</file>
    <file subId="24">chrY.fna</file>
    <file subId="25"
uriPath="ftp://ftp.ncbi.nlm.nih.gov/genomes/all/GCA_000001405.21_GRCh38.p6/GCA_000001405.21_GR
Ch38.p6_assembly_structure/non-nuclear/assembled_chromosomes/FASTA">chrMT.fna</file>
  </dataIn>

  <dataOut>
    <path>/dss004/BulkData/R/NCBI/GRCh38.p6</path>
  </dataOut>

</AriocE>
```

Configuration file for AriocE (reference sequences) for a nongapped (spaced-seed) hash table.

To build the corresponding gapped (spaced-seed) hash tables, change the parameters specified in the XML attributes of the `<AriocE>` element and re-execute AriocE:

```
<AriocE seed="hsi20_0_30" maxDOP="32" maxJ="200">
```

Here is a list of additional configuration-file parameters for reference-sequence encoding:

Element	Attribute	Description
<AriocE>	maxDOP	Number of concurrent CPU threads
	maxJ	Maximum hash table bucket size ("big bucket" threshold)
	seed	One of the following values: ssi84_2_29 (seed width 84, 2 mismatches, 29-bit hash) ssi84_2_30 (seed width 84, 2 mismatches, 30-bit hash) hsi20_0_29 (seed width 20, 0 mismatches, 29-bit hash) hsi20_0_30 (seed width 20, 0 mismatches, 30-bit hash)
<dataIn>	srcId	Data source ID (between 0 and 4,194,303)
	filePath	Parent directory for files specified in <file> elements
	uriPath	Original URI for reference sequences (copied to the UR field in the @SQ records in SAM-formatted output)

Optional elements and attributes in the AriocE configuration file for encoding reference sequences or a genome.

Most of these parameters are self-evident, but the seed= and maxJ= values deserve careful attention.

seed=. The seed= value for the nongapped hash table should typically be ssi84_2_30 (seed width 84, up to 2 mismatches, 30-bit hash values); for the gapped hash table, the value should typically be hsi20_0_30 (seed width 20, no mismatches, 30-bit hash values). It is possible to trade memory space for speed: with 29-bit hash values, the aligners use only about 36 gigabytes of memory, but alignment throughput decreases because of the need to resolve more hash-table collisions.

maxJ=. The maxJ= parameter trades speed for sensitivity. For the nongapped hash table, this effect is barely noticeable. In this case, you may simply omit the maxJ= parameter so that the maximum size of a hash table bucket is unlimited.

For the gapped hash table, however, maximum throughput is obtained with a value of about 16 (but with less sensitivity), whereas maximum sensitivity can be achieved by using values of 250 and up (but with a corresponding decrease in throughput). When encoding a genome such as the human genome that contains numerous repetitive regions, you should always specify a value for maxJ=. This causes AriocE to optimize the hash table by pruning repetitive regions of the genome where adjacent seeds hash to "big buckets"; this optimization improves throughput significantly with very little decrease in sensitivity for reads that map to repetitive regions.

AriocU: aligning unpaired reads

Before running AriocU, use AriocE to encode a reference sequence and hash tables (see AriocE: encoding sequence data, page 11).

To run AriocU, use a configuration file that specifies where to find encoded reference-sequence and read files, where to write alignment-result files, and how to parameterize the aligner:

```
<?xml version="1.0" encoding="utf-8"?>

<AriocU gpuMask="0x00000001" batchSize="64k" maxDOP="24">

  <R>C:\BulkData\R\NCBI\GRCh38.p6</R>

  <nongapped seed="ssi84_2_30" />

  <gapped seed="hsi20_0_30" Wmxgs="2_6_5_3" Vt="400" />

  <Q filePath="C:\BulkData\Q\mason\test203">
    <unpaired subId="1">
      <file>i250p_1</file>
    </unpaired>
  </Q>

  <A overwrite="true">
    <sam report="mu">C:\BulkData\A\mason</sam>
  </A>

</AriocU>
```

Simple configuration file for AriocU.

Each of the XML elements in the configuration file controls a different aspect of the read-alignment pipeline:

- `<AriocU>`: GPU devices, GPU memory, and CPU threads
- `<R>`: encoded reference sequence files
- `<nongapped>`: seed type for nongapped alignments
- `<gapped>`: seed type and scoring parameters for gapped alignments
- `<Q>`: encoded read files
- `<A>`: output files and formats

The following table summarizes the basic configuration parameters:

Element	Attribute	Description
<AriocU>	gpuMask	Bits corresponding to GPU devices
	batchSize	Number of reads per batch per GPU
	maxDOP	Number of concurrent CPU threads
<nongapped>	seed	Seed used for AriocE nongapped reference-sequence encoding
	maxA	Maximum number of reported nongapped mappings per read
<gapped>	seed	Seed used for AriocE gapped reference-sequence encoding
	Wmxgs	Smith-Waterman scoring parameters
	Vt	Minimum reportable alignment score
	maxA	Maximum number of reported gapped mappings per read
<Q>	maxQ	Limits the number of reads aligned (see)
<unpaired>	subId	Subunit ID used for AriocE read encoding
<A>	overwrite	"true" to overwrite existing result file(s)
	mapqVersion	0: similar to BWA-MEM 1: similar to Bowtie 2 2: modified Bowtie 2
<sam>, <sbf>	report	Alignment result type(s) to report in SAM or SBF format in the specified output directory

XML elements and attributes in the AriocU configuration file.

Some of these configuration parameter values are self-evident, but others demand careful attention:

gpuMask=. The gpuMask= parameter is a bit mask (specified as a hexadecimal value) that indicates which GPU device(s) are to be used by AriocU. Each bit in the parameter corresponds to a GPU device identifier. If there is only one GPU in the computer, the value of gpuMask= should be 1. To run AriocU on two or more GPUs, set the corresponding bits in gpuMask=.

batchSize=. The batchSize= parameter determines how many reads AriocU can process concurrently per GPU. Batch sizes are limited by available GPU memory, but the amount of GPU memory needed to compute alignments cannot be precisely predicted because it depends on the number of Smith-Waterman computations AriocU must perform in its search for satisfactory mappings for every read — and this is something you don't know until you try it.

With some data, we have observed overall speed increases on the order of 10% with "optimally large" batch sizes, but batch size has less effect on speed than other "tuning" parameters (see Tuning for speed and sensitivity, page 27). If you do want to try to find an optimal batch size for your data, we suggest that you start with a batch size of about 32k. If AriocU runs without memory-allocation errors, progressively increase the batch size and re-run AriocU until it runs out of GPU memory or until its throughput does not improve with increasing batch size.

The batchSize= parameter may be specified either as a positive integer (e.g., "25000") or as a multiple of 1024 (e.g., "128K").

maxDOP=. AriocU uses CPU threads for file input/output and for post-processing alignment data (computing mapping qualities, formatting alignment results, and so on). AriocU's throughput is

generally limited by GPU speed rather than CPU speed, but you should nevertheless let AriocU use as many CPU threads as you have available.

Wmxgs=. This parameter specifies four Smith-Waterman scoring parameters as non-negative integers separated by commas or underscores. The four values indicate the score for a match, mismatch, gap open, and gap space respectively. For example, "2,6,5,3" means match=+2; mismatch=−6; gap open=−5; gap space=−3.

Vt=. This parameter indicates the minimum reportable Smith-Waterman alignment score. You can specify `Vt=` in one of two ways: as a non-negative integer threshold score (e.g., "100") or as a function of read length described in a Bowtie-style function parameter: "[LSG],b,a", where the initial letter indicates the function type (L: linear; S: square root; G: natural log), b is a constant term and a is the coefficient. For example, `Vt="S,75,2.5"` specifies a reportable alignment score threshold of `75 + 2.5*sqrt(read length)`.

mapQversion=. The `mapqVersion=` parameter specifies the computational model to use when reporting mapping qualities (MAPQ) for reads. AriocU can compute MAPQ using a model similar to either BWA-MEM (numerical range 0-60) or Bowtie 2 (numerical range 0-44).

report=. Use the `report=` parameter to filter reported alignment results. You can use any combination of `m` (mapped) and `u` (unmapped). For example, to report both mapped and unmapped results in the same SAM file, use a `<sam>` element with `report="mu"`. To report the same results to two separate SAM files, use two `<sam>` elements, one with `report="m"` and one with `report="u"`.

AriocP: aligning paired-end reads

You must use AriocE to encode a reference sequence and hash tables (see AriocE: encoding sequence data, page 11) before you run AriocP.

The AriocP configuration file is similar to the one that AriocU uses. The important differences have to do with parameters that control the alignment of paired-end reads:

```
<?xml version="1.0" encoding="utf-8"?>

<AriocP gpuMask="0x00000001" batchSize="48k" maxDOP="24">

  <R>/BulkData/R/NCBI/GRCh38.p6</R>

  <nongapped seed="ssi84_2_30" />

  <gapped seed="hsi20_0_30" Wmxgs="2,6,5,3" Vt="100" />

  <Q filePath="/BulkData/Q/mason/test103">
    <paired subId="1">
      <file>i100p_1</file>
      <file>i100p_2</file>
    </paired>
  </Q>

  <A overwrite="true" pairOrientation="c" pairCollision="oc">
    <sam report="cdru">/BulkData/A/mason</sam>
  </A>

</AriocP>
```

Simple configuration file for AriocP.

Each of the XML elements in the configuration file controls a different aspect of the read-alignment pipeline:

- `<AriocP>`: GPU devices, GPU memory, and CPU threads
- `<R>`: encoded reference sequence files
- `<nongapped>`: seed type for nongapped alignments
- `<gapped>`: seed type and scoring parameters for gapped alignments
- `<Q>`: encoded read files
- `<A>`: output files and formats

The following table summarizes the basic configuration parameters:

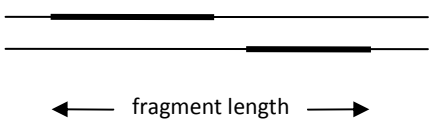
Element	Attribute	Description
<AriocP>	gpuMask	Bits corresponding to GPU devices
	batchSize	Number of reads per batch per GPU
	maxDOP	Number of concurrent CPU threads
<nongapped>	seed	Seed used for AriocE nongapped reference-sequence encoding
	maxA	Maximum number of reported nongapped mappings per read
<gapped>	seed	Seed used for AriocE gapped reference-sequence encoding
	Wmxgs	Smith-Waterman scoring parameters
	Vt	Minimum reportable alignment score
	maxA	Maximum number of reported gapped mappings per read
<Q>	maxQ	Limits the number of reads aligned (see Tuning for speed and sensitivity, page 27)
<paired>	subId	Subunit ID used for AriocE read encoding
<A>	overwrite	"true" to overwrite existing result file(s)
	pairFragmentLength	Distance between the ends of a paired-end mapping
	pairOrientation	Specifies the relative orientation (convergent/divergent/same) of the mates in a pair.
	pairCollision	Indicates whether mates may overlap, cover, or dovetail.
	mapqVersion	0: similar to BWA-MEM 1: similar to Bowtie 2 2: modified Bowtie 2
<sam>, <sbf>	report	Alignment result type(s) to report in SAM or SBF format in the specified output directory

XML elements and attributes in the AriocP configuration file.

Most of the AriocP configuration parameters control the same behaviors as they do with AriocU. The important differences are related to the management of paired-end alignments:

Vt=. The Vt= parameter is the minimum reportable alignment score for each individual mate in a pair, not the combined alignment scores for both mates.

pairFragmentLength=. The pairFragmentLength= parameter specifies a range of acceptable fragment-length values. AriocP computes the pair fragment length as a SAM TLEN, that is, the number of mapped reference-sequence positions between the first ("upstream") mapped base and the last ("downstream") mapped base in a pair of mappings:

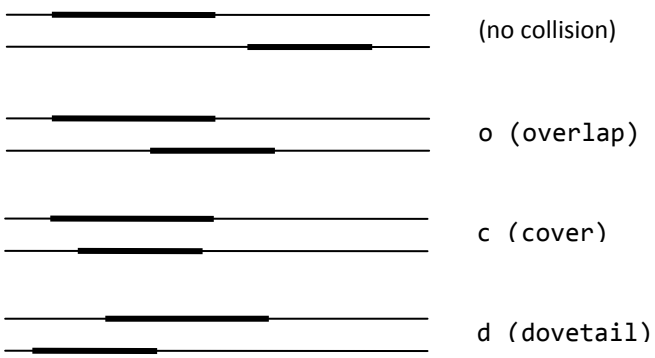


pairOrientation=. This parameter specifies the expected orientation of the mates in a pair:

- convergent: the "upstream" mate (that is, the mate at the lower-numbered location on the reference sequence) maps to the forward strand and the opposite mate maps to the reverse-complement strand

- divergent: the upstream mate maps to the reverse-complement strand and the opposite mate to the forward strand
- same: both mates map to the same strand

pairCollision=. The pairCollision= parameter indicates whether the mappings of a pair of mates may overlap, cover, or dovetail each other:



report=. The report= parameter value can be any combination of the following (similar to Bowtie's paired-end alignment result categories):

- c (concordant): both mates in a pair have reportable mappings and meet the user-specified constraints for orientation and fragment length
- d (discordant): each mate has a unique mapping, but the mappings do not meet the user-specified paired-end constraints
- r (rejected): both mates have mappings and at least one of the mates has two or more mappings, but no combination of the mappings meets the user-specified paired-end constraints
- u (unmapped): one or both of the mates has no reportable mapping

You can write alignment results to two, three, or four different files by using multiple <sam> elements, each with a different, mutually-exclusive subset of the four alignment-result categories.

Alignment results

Both AriocU and AriocP can generate alignment results in two different formats: SAM [SAM/BAM Format Specification Working Group, 2014] and SBF (the binary format that the Microsoft SQL Server uses for "native" bulk inserts). The `<sam>` and `<sbf>` elements in the configuration file control whether the Arioc aligners emit results in SAM, SBF, or both formats.

[The Arioc aligners can also produce alignment results in the binary format used for data import into the Terabase Search Engine database. See the Arioc source code for details.]

SAM results

The Arioc aligners emit the following tab-separated fields. (See the SAM specification document for details):

	Field name	Description
1	QNAME	Query-sequence (read) metadata
2	FLAG	Flag bits
3	RNAME	Reference sequence name
4	POS	Mapping position
5	MAPQ	Mapping quality
6	CIGAR	CIGAR string
7	RNEXT	Opposite mate reference sequence name
8	PNEXT	Opposite mate mapping position
9	TLEN	Paired-end fragment length
10	SEQ	Query sequence
11	QUAL	Phred-scaled base quality scores
12	id	Unique sequence identifier
13	AS	Alignment score
14	XS	Second-highest alignment score
15	YS	Opposite mate alignment score
16	NM	Edit distance
17	MD	MD string
18	YT	Alignment result category
19	NA	Number of distinct mappings found by the aligner
20	NB	Number of maximum-score Smith-Waterman mappings
21	RG	Read group
22	af	Aligner flags
23	qf	Query flags

Tab-separated fields in Arioc SAM files.

Here are some additional details about the SAM fields reported by AriocU and AriocP:

`RNAME`: The Arioc aligners use a 3-digit decimal string in `RNAME`. This numerical identifier corresponds to the `subId=` parameter that you associate with each reference sequence at the time the sequence is encoded (see Encoding a reference sequence, page 14). The metadata associated with each reference

sequence (that is, in the original FASTA file for the reference sequence) appears in the `rm=` field of the corresponding `@SQ` record in the SAM file.

`id`: The AriocE encoder assigns a unique 64-bit sequence identifier to each sequence it reads from a FASTA or FASTQ file. The AriocU and AriocP aligners emit this unique identifier in the `id` field, but they do not preserve numerical ordering of the identifiers.

`YT`: AriocP assigns a 2-character "mapping type" (based on the YT field in Bowtie 2) to each SAM record:

YT	Description
CP	Concordant (meets user-specified criteria for mate orientation and fragment length)
DP	Discordant (unique mapping that does not meet user-specified criteria for orientation and fragment length)
RP	Rejected (both mates mapped without meeting user-specified criteria for orientation and fragment length, and one or both mates had multiple mappings)
UP	Unmapped (one or both mates unmapped)

Paired-end mapping types reported by AriocP.

`NA`: Because the Arioc aligners evaluate potential alignments in parallel, the number of mappings they discover may be significantly greater than the number of mappings they report. The NA field indicates the total number of distinct mappings that were found by the aligner.

`NB`: In the scoring matrix computed for Smith-Waterman alignment, two or more different cells may contain the same maximum alignment score. The `NB` field records the number of different cells that contained the same maximum score for the reported mapping.

`af, qf`: These fields are used internally by the Arioc aligners.

SBF results

The binary format of SBF files generated by the Arioc aligners corresponds to the following SQL table definition (where N is a placeholder for a user-specified maximum size for variable-length strings):

```
create table AriocSBFTable
( qname    varchar(N)      not null,
  flag     smallint        not null,
  rname    smallint        not null,
  pos      int             not null,
  mapq     tinyint         not null,
  cigar    varchar(N)      not null,
  rnext    smallint        not null,
  pnext    int             not null,
  tlen     int             not null,
  seq      varchar(N)      not null,
  qual     varbinary(N)    not null,
  sqId     bigint          not null,
  V        smallint        not null,
  XS       smallint        not null,
  NM       smallint        not null,
  MD       varchar(N)      not null,
  NA       smallint        not null,
  NB       smallint        not null,
  RG       smallint        not null,
  af       smallint        not null,
  qf       smallint        not null
)
```

SQL table definition corresponding to the Arioc SBF file format.

Tuning for speed and sensitivity

Arioc supports a variety of parameters you can use to control its behavior and balance between speed and sensitivity.

The verboseMask= and maxQ= parameters

The verboseMask= parameter (used with the <AriocE>, <AriocU>, or <AriocP> element) controls the amount of detail in the output of each of the Arioc executables. Specify the parameter value as a 32-bit hexadecimal bitmap:

Bit	Hexadecimal	Description
31	0x80000000	Output to stderr
30	0x40000000	Output to Windows debugger (OutputDebugString)
29	0x20000000	Emit timestamps
11	0x00000800	Trace main loop iterations
9	0x00000200	Trace CUDA (GPU) memory management
8	0x00000100	Trace host (CPU) memory management
4	0x00000010	Detailed trace
3	0x00000008	Basic trace
2	0x00000004	Detailed performance metrics
1	0x00000002	Basic performance metrics
0	0x00000001	Banners and exceptions

Bit flags in the verboseMask= parameter.

For performance tuning, use verboseMask=0xE0000807 to see time-stamped intermediate results and summary timings.

The maxQ= parameter (used with the <Q> element) halts read alignment after the specified number of reads (for AriocU) or pairs (for AriocP) have been processed per GPU.

Tuning for sensitivity

Arioc finds more high-scoring mappings when it evaluates potential alignments at more reference-sequence locations. Here are several ways to broaden Arioc's "search space":

AriocE: Build the seed-and-extend hash table with a higher value for maxJ=. This affects "big bucket" hash values, that is, seeds that derive from highly-repetitive regions and that consequently are associated with many reference-sequence locations. Increasing maxJ= thus causes AriocU and AriocP to compute more potential alignments for reads that contain "big bucket" seeds.

Increasing maxJ= has no effect on most hash table buckets because seeds that are not associated with highly-repetitive regions of the reference hash to very few reference-sequence locations anyway. Thus, with increased maxJ=, AriocU and AriocP find additional mappings primarily for a few noisy reads. (By "noisy", we mean that a read contains enough mismatches and indels that only a few of its seeds remain intact. When those intact seeds happen to be associated with "big buckets" in the hash table, the aligner is more likely to find a mapping when you use a larger value for maxJ=.)

With larger values for `maxJ=`, AriocU and AriocP spend more time evaluating additional seed locations. With the human reference genome, speed starts to decrease sharply with `maxJ=` values above 500 or 1000, with relatively small improvements in sensitivity.

AriocU, AriocP: To obtain greater sensitivity from the AriocU and AriocP aligners, you can increase the value of any or all of the following parameters:

Element	Attribute	Description
`<gapped>`	`AtN`	Maximum number of nongapped mappings per read
	`seedDepth`	Number of seed iterations
	`AtG`	Maximum number of gapped mappings per read

Configuration parameters useful for AriocU and AriocP performance tuning.

To understand how to use these parameters, consider how the aligners process a read.

First, the aligner uses periodic spaced seeds to find nongapped mappings that contain no more than two mismatches. The value of `AtN=` is a threshold for the number of nongapped mappings; if the aligner finds at least that number of nongapped mappings for a read, it excludes the read from further processing by the gapped aligner. This parameter setting makes a difference for the occasional read whose best nongapped mapping has a lower alignment score than the read's best gapped mapping. You can find the higher-scoring gapped mappings for such reads by setting `AtN=` to a value of 2 or more.

After searching for nongapped mappings, the aligner uses seed-and-extend to find gapped mappings for the remaining reads (i.e., those that do not have the minimum number of nongapped mappings). For each such read, the aligner iteratively chooses seeds, starting with the set of seeds that are immediately adjacent to each other. (For example, with 20nt seeds, the aligner starts with the seeds at positions 0, 20, 40... in the read). If none of these seeds leads to a successful mapping, the seed interval is halved and the new set of seeds is evaluated. With typical seed sizes, six such "interval halving" iterations are needed to evaluate every possible seed location in the read. You can alter this behavior by setting the `seedDepth=` parameter, which limits the number of seed iterations to a number between 1 and 5. The Arioc aligners run significantly faster with smaller values of `seedDepth=`, but they also find high-scoring mappings for fewer reads.

Arioc abandons the search for seed-and-extend mappings for a read when it finds a sufficient number of high-scoring mappings. You can use the `AtG=` parameter to force Arioc to keep searching for higher-scoring mappings; the aligner will not give up on a read until it finds at least the number of gapped mappings specified in `AtG=` (or until it runs out of seeds).

Because Arioc computes potential alignments in parallel, neither `AtN=` nor `AtG=` is precisely deterministic. For example, when you set `AtN=` to 2, Arioc might nevertheless find 10 high-scoring nongapped mappings for a particular read, simply because all 10 mappings were discovered in parallel.

Tuning for speed

Arioc runs faster when it searches for fewer mappings. In general, choosing lower values for the same set of tuning parameters leads to higher throughput.

AriocE: Build the seed-and-extend hash table with a lower value for `maxJ=`. This causes AriocE to be more aggressive in identifying "big bucket" regions of the reference genome and pruning the hash table accordingly. You can use values for `maxJ=` as low as 16 and still obtain acceptable alignment results with AriocU and AriocP.

AriocU, AriocP: The AriocU and AriocP aligners also recognize the `maxJ=` parameter. (Use `maxJ=` on the `<nongapped>` and `<gapped>` elements in the configuration file.) In the aligners, `maxJ=` specifies a limit to the number of reference-sequence locations that the aligners examine for each seed. Changing `maxJ=` has little effect when used with the nongapped aligner (which is very fast anyway) but using lower values for `maxJ=` can tangibly increase the speed of the gapped aligner.

You can also increase throughput by using smaller values for the `seedDepth=`, `AtN=`, and `AtG=` parameters. As you decrease these parameter settings, you may notice that speed is limited by factors other than the intrinsic speed of the aligners. In particular, disk I/O bandwidth and contention for the PCIe bus can each limit overall throughput, especially when you run AriocU or AriocP on multiple GPUs.

References

Altschul SF et al. (1990) Basic Local Alignment Search Tool. J Mol Biol **215**, 403-410.

Cooper A. (2004) The inmates are running the asylum. Sams Publishing, Indianapolis, IN. ISBN 0-672-32614-0.

Illumina Corporation. (2015) CASAVA v1.8 User Guide. Available at http://support.illumina.com/help/SequencingAnalysisWorkflow/Content/Vault/Informatics/Sequencing_Analysis/CASAVA/swSEQ_mCA_FASTQFiles.htm.

Liu Y et al. (2013) CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. BMC Bioinformatics **14**:117.

SAM/BAM Format Specification Working Group. (2014) Sequence Alignment/Map Format Specification. Available at https://github.com/samtools/hts-specs.

Smith TF, Waterman MS. (1981) Identification of common molecular subsequences. J Mol Biol **147**, 195-197.

Ukkonen E. (1983) On approximate string matching. Springer: Foundations of Computer Theory, Lecture Notes on Computer Science **158**, 487–495.

Wilton R et al. (2015) Arioc: high-throughput read alignment with GPU-accelerated exploration of the seed-and-extend search space. PeerJ **3**:e808; DOI 10.7717/peerj.808