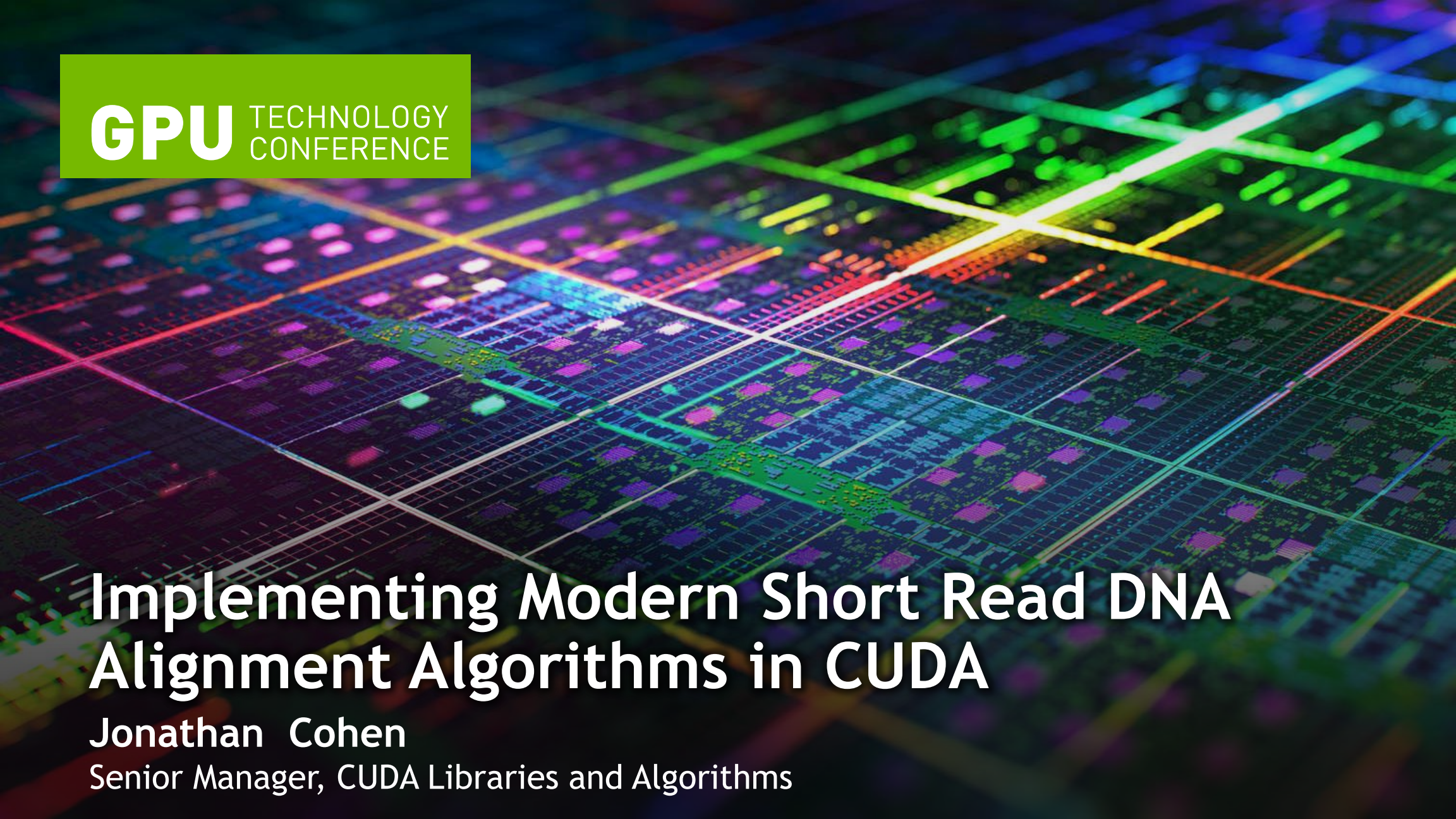
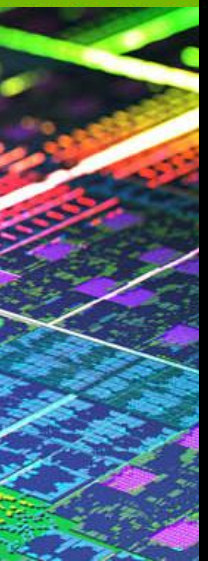


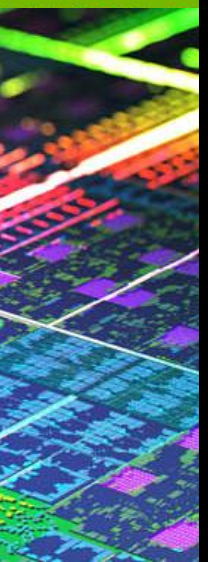
Implementing Modern Short Read DNA Alignment Algorithms in CUDA

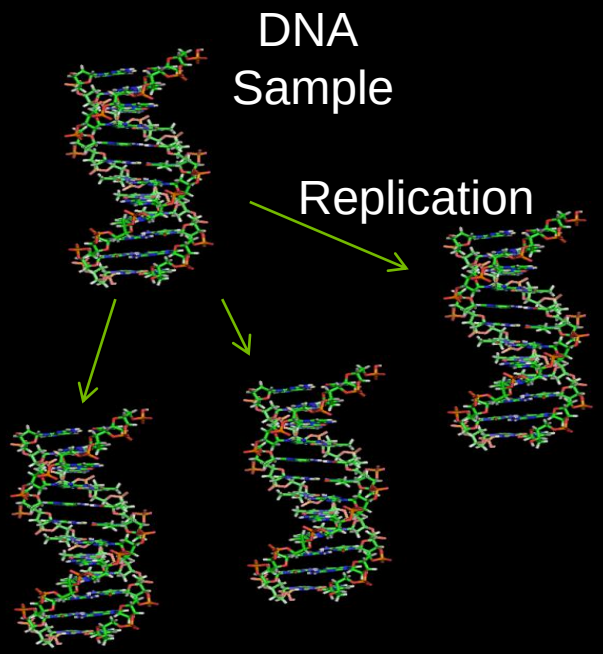
Jonathan Cohen
Senior Manager, CUDA Libraries and Algorithms



Next-Gen DNA Sequencing

- In 4 slides





C

A

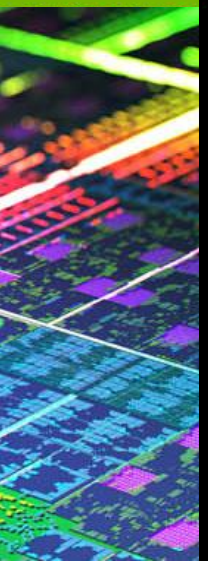
T

G

Sensing Circuitry

G C

...



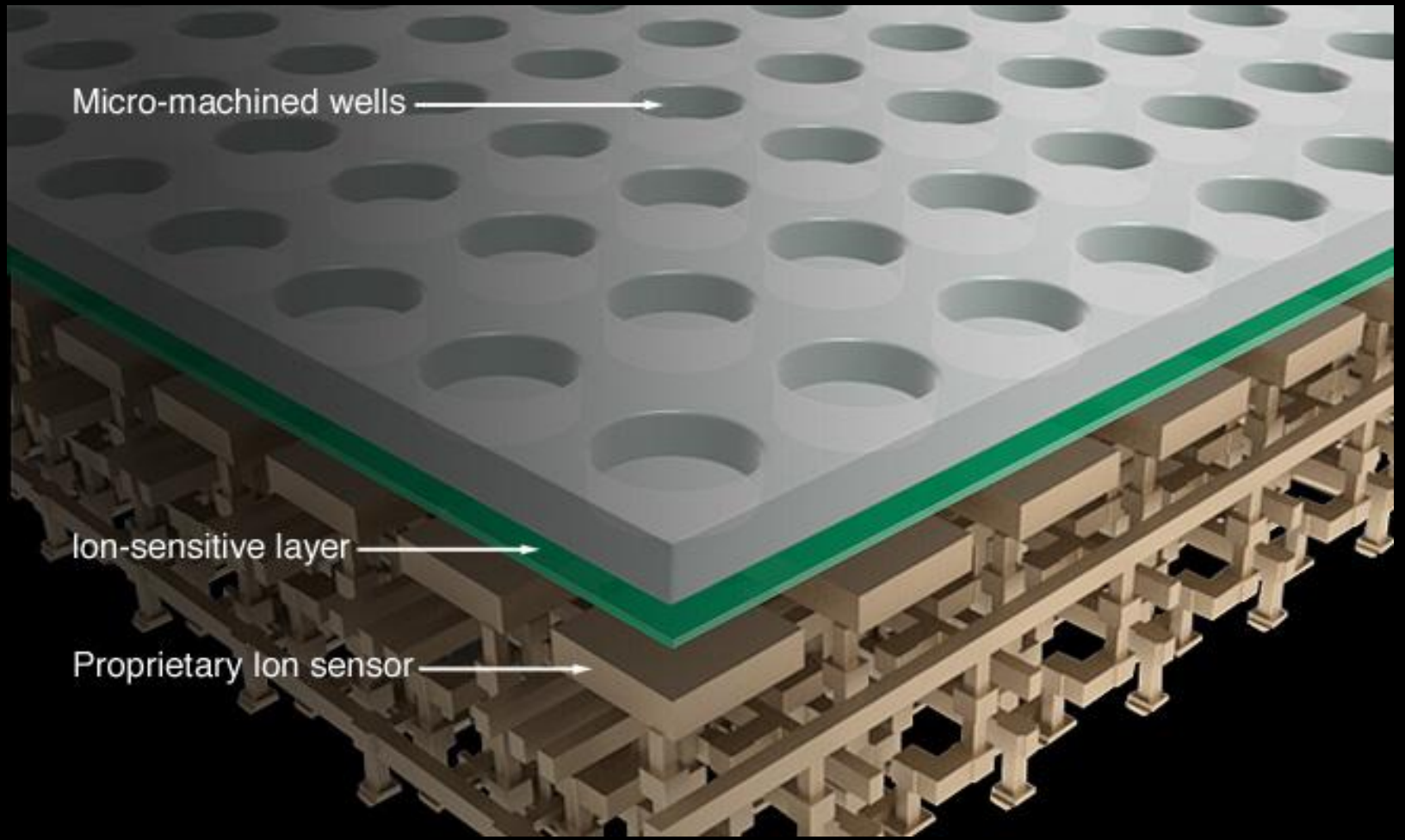


Image courtesy Life Technologies

Inside the Micro Well

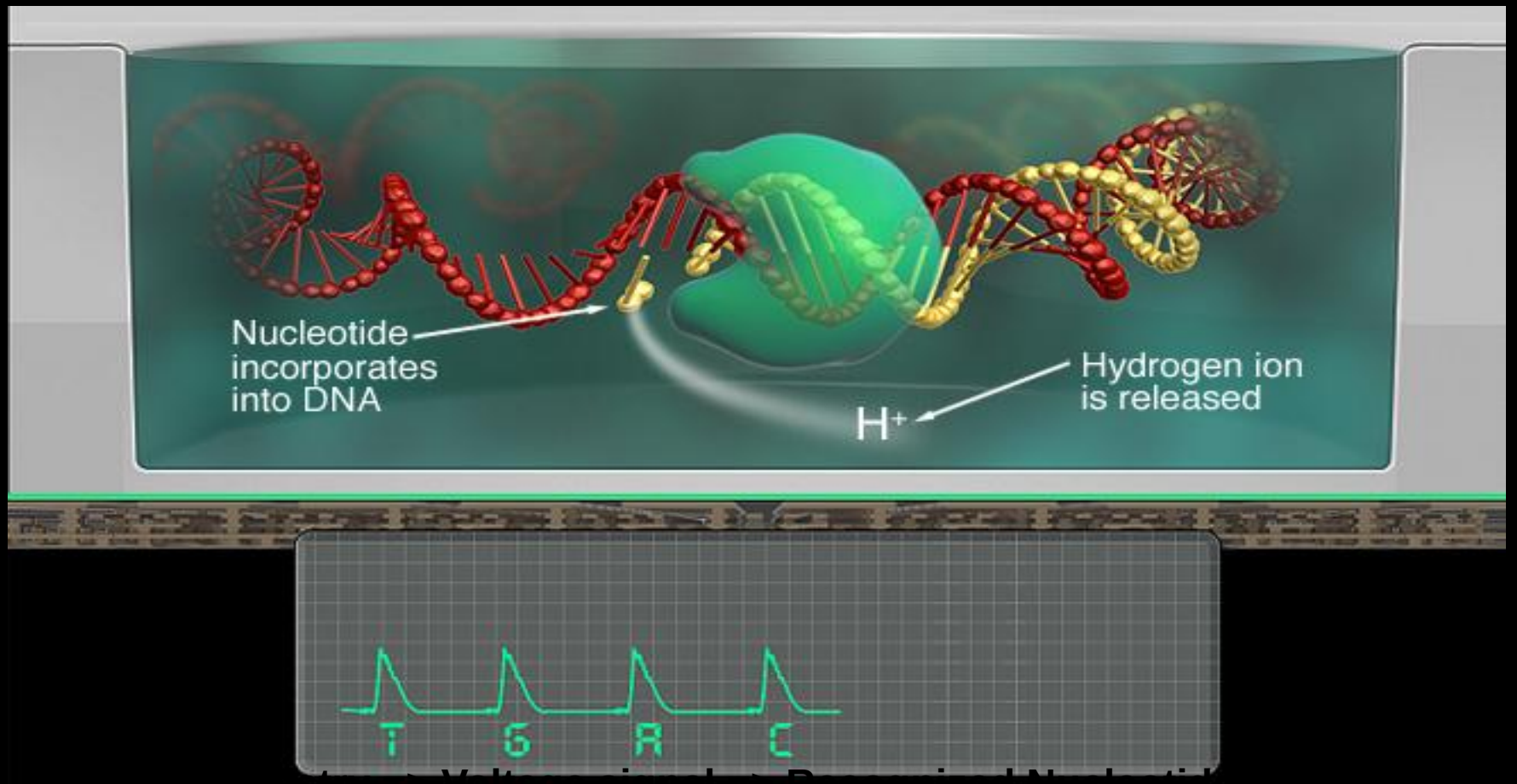


Image courtesy Life Technologies

Short Reads – Mapped to Reference

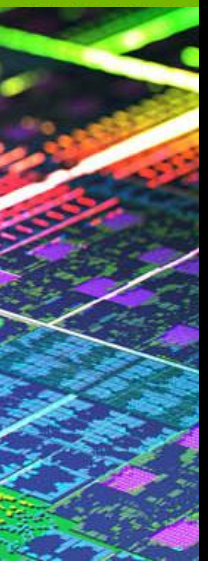
GACTAGATAAGGATACA

ATAAGGATACAATTAGCG

GGCGACAGAGGTAAGA

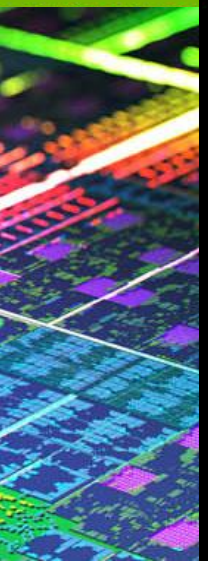
GTCGAGATTAAGTGGC

CATAGATTGACAAATGACCAGATAAGGATACAATTAGCGGCCACAGAGGCGAGATTAAGTGGCAGATAG



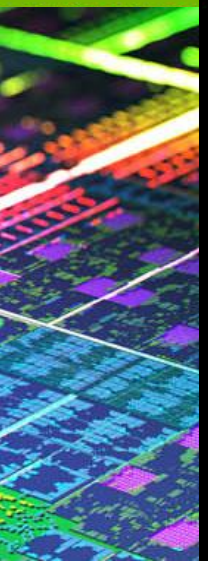
Critical Data Structure: FM-Index

- Query: “Find all substrings beginning with ‘X’ in reference”
- Incremental Query: “Given all substrings beginning with ‘X’, find which begin with ‘XY’”
- Build substring search results letter by letter
- Each incremental query is O(1)



Modern Mapping Algorithm - BWA

- BWA – bounded 9-ary exponential search
- To match: CACCT, 1 allowed mismatch
  - [Match] Search for C…, recurse on “ACCT”, 1 allowed mismatch
  - [C->A] Search for A…, recurse on “ACCT”, 0 allowed mismatches
  - [C->T] Search for T…, recurse on “ACCT”, 0 allowed mismatches
  - [C->G] Search for G…, recurse on “ACCT”, 0 allowed mismatches
  - [Del] recurse on “ACCT”, 0 allowed mismatches
  - [Ins. A] Search for A…, recurse on “CACCT”, 0 allowed mismatches
  - …
  - [Ins. C] Search for C…, recurse on “CACCT”, 0 allowed mismatches



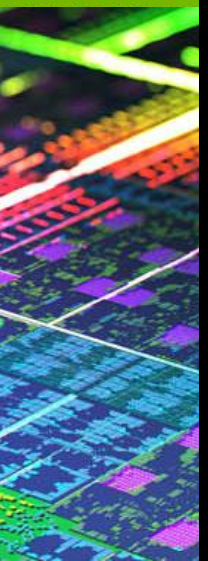
Modern Mapping Algorithms - Bowtie2

- Seed and extend
  - find a small matching “seed” substring (allowing some errors, similar to BWA)
  - extend in either direction with Smith Waterman (or Edit Distance)
- Either:
  - Use heuristics to only extend “promising seeds” (best mapping)
  - Extend all seeds, guaranteed to find all matches within allowed error (all mapping)
- The Problem with Both Approaches: **Branchy Code!**

Branchy Code on GPUS

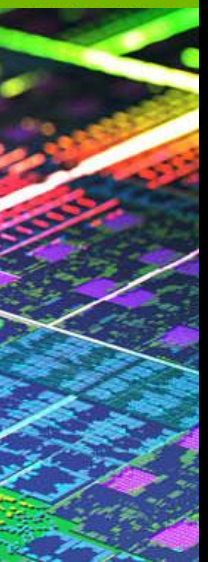
- GPUs have tons of bandwidth and ops/sec
- But they are wide SIMT…
- => perf ~ utilization
- And branchy code often means low utilization…
- => Read mapping algorithms not fit for GPUs ?

FALSE!



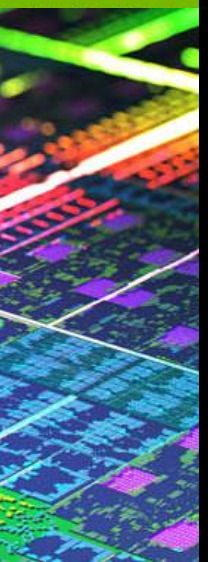
Branchy code and GPUs

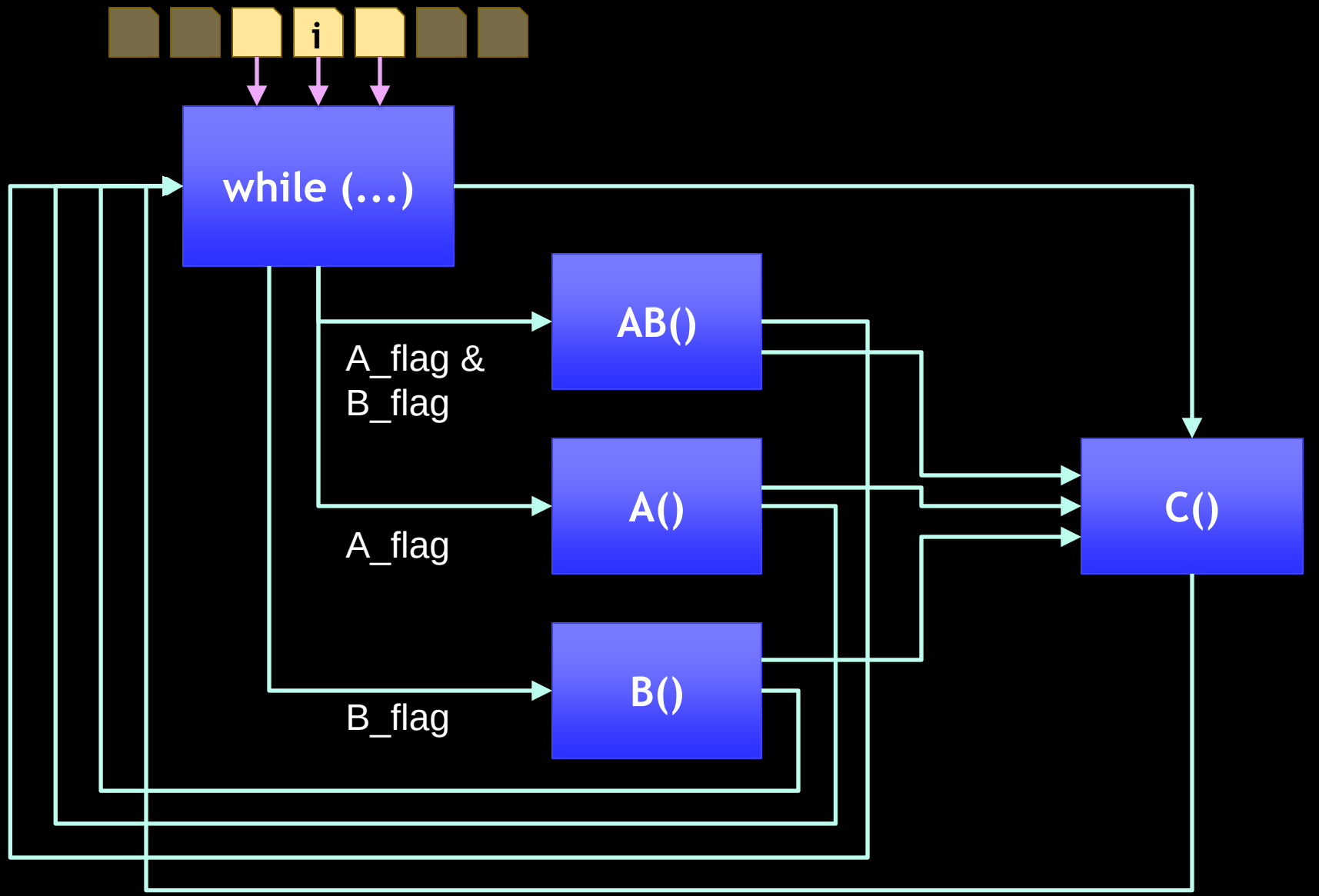
- Stop thinking serially
- think of your code as a **pipeline** whose **stages** are formed by the **bodies** of your branches
- and embrace the concept of **pipeline parallelism**



An Example

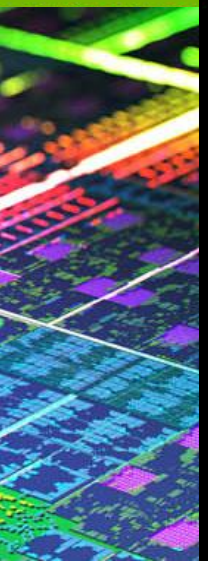
```
void process(int i)
{
    State state(i);
    while (state.is_done() == false)      // taken 30% of the times
    {
        if (state.A_flag)                 // taken 50% of the times
        {                                 //    => 15% utilization!
            if (state.B_flag)             // taken 33% of the times
                AB( state );              //    => 5% utilization!
            else
                A( state );               //    => 10% utilization!
        }
        else if (state.B_flag)            // taken 50% of the times
            B( state );                   //    => 15% utilization!

        if (state.C_flag)                 // taken 33% of the times
            C( state );                   //    => 10% utilization!
    }
}
```





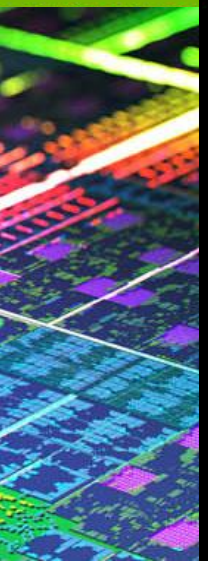
Pipeline Parallelism

- Work flows through queues in the form of packets
- State can either flow with the work packets or be dereferenced (or both)
- At the entry point of each stage utilization is 100%



An Example, Revisited

```
__host__ bool pipeline() { // pipeline scheduler - CPU
    if (while_q.size() > thresh) while_stage<<<while_q.size()>>>();
    if (AB_q.size() > thresh) AB_stage<<<AB_q.size()>>>(); // etc.
    return while_q.empty() && AB.empty() && ...;
}

__global__ void while_stage() { // primary stage - GPU
    const int tid = thread_id();        // thread id
    if (tid >= in_queue.size()) return;

    const State state = in_queue[tid]; // fetch work from input queue
    if (state.A_flag)
    {
        if (state.B_flag) AB_queue.push( state );
        else              A_queue.push( state );
    }
    else if (state.B_flag) B_queue.push( state );
    else if (state.C_flag) C_queue.push( state );
}
```



Pros

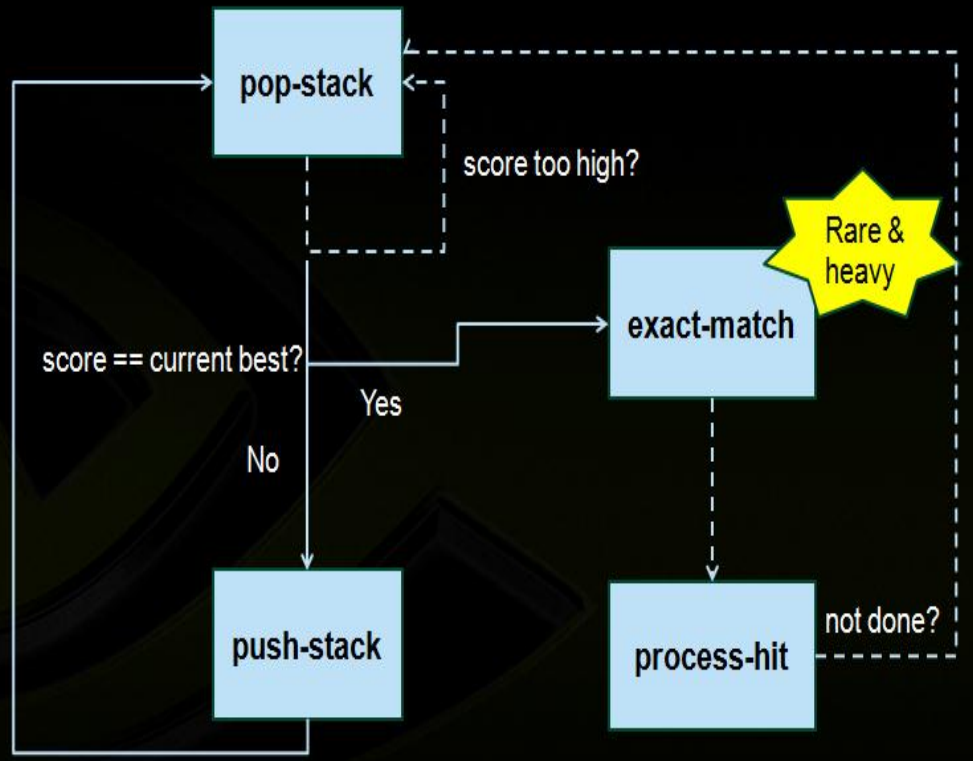
- At the entry point of each stage utilization is 100%
- each stage a kernel => reduced register pressure, higher occupancy
- Before launching work from a queue, the work packets can be sorted for better memory coherence

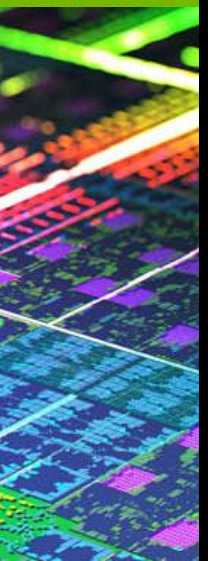
Cons

- Some additional memory traffic, but mostly coherent

nvBWA

- Proof-of-concept using this concept
- C2075: 3-4x vs. BWA on a 6-core SNB, 4-5x vs. BarraCUDA





nvBowtie2

- From-scratch implementation of Bowtie2 algorithm
- Transformed using pipeline parallelism approach
- Many other system-level of algorithm optimizations
- Goal is to produce the same results (or statistically identical results)

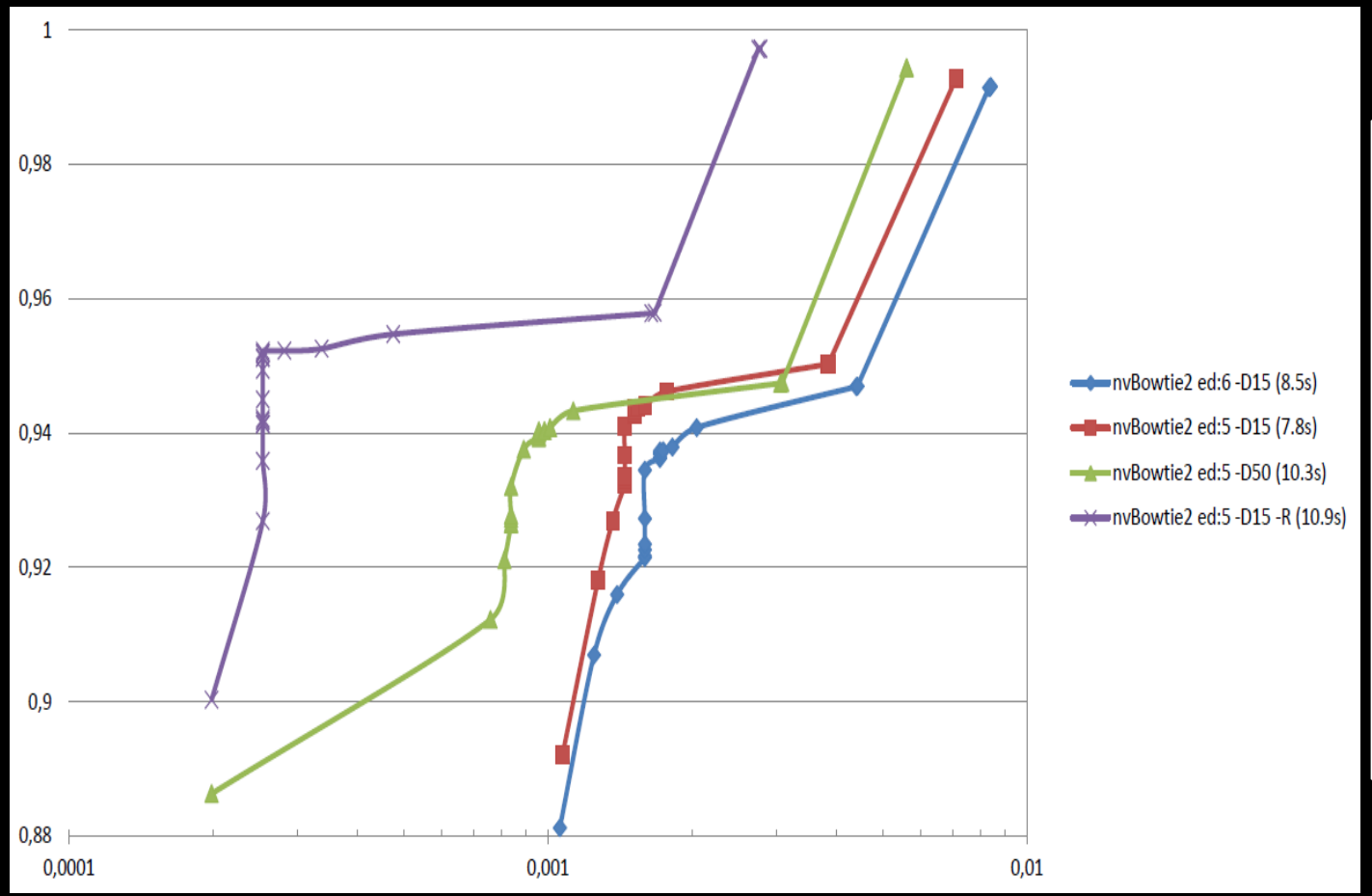
Best-mapping

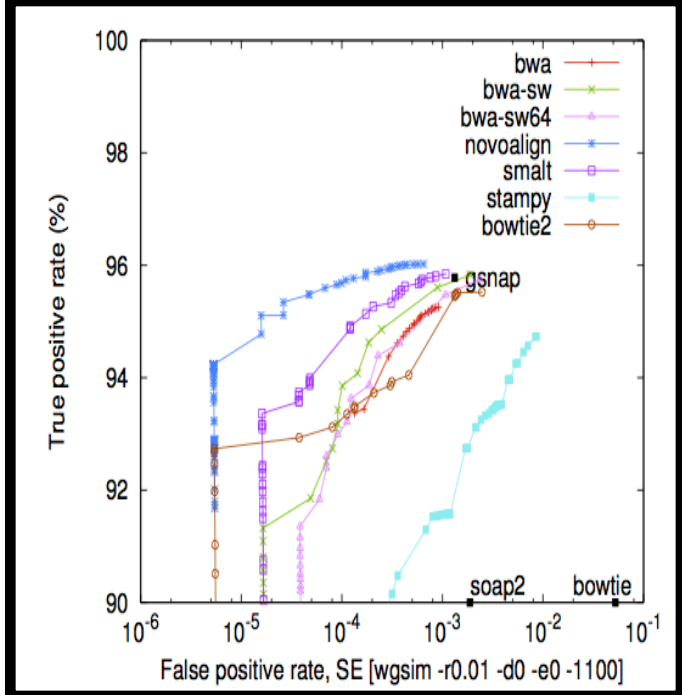
- ERR012100 dataset, 10M x 100bp reads

Software	Hardware	Time
Bowtie2 (SW)	Xeon X5650, 1 core	57m:41s
Bowtie2 (SW)	Sandybridge, 6 core (est.)	285s (best-case estimated)
nvBowtie2 (SW)	K20C	65.1s
Masai (ED=5)	Xeon X5650, 1 core	24m.56s
Masai (ED=5)	Sandybridge, 6 core (est.)	125s (best-case estimated)
nvBowtie2 (ED=5)	K20C	48.5s

- (Masai uses a superior algorithm, we are working on a port)

nvBowtie2: wgsim (1M x 100bp)





Best Mapping – longer reads

- SRR493095 dataset, 857K x 150bp reads

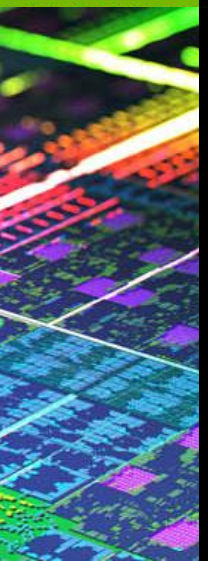
Software	Hardware	Time
Bowtie2	Core i7-3930K (12 threads)	57.4s
nvBowtie2	K20C	11.8s

All Mapping

- ERR012100 dataset, 10M x 100bp reads

Software	Hardware	Time	Best-case Sandybridge (est.)
RazerS3	Xeon X5650 (1 core)	3653m:03s	304m:26s
Hobbes	Xeon X5650 (1 core)	2319m:27s	193m:17s
mrFAST	Xeon X5650 (1 core)	4462m:25s	371m:52s
Masai (ED=5)	Xeon X5650, 1 core	284m:34s	23m:43s
nvBowtie2 (ED=5)	K20C	31m:7s	

- (Masai uses a superior algorithm, we are working on a port)



Questions?

- Most of this work from Jacopo Pantaleoni

- Questions to:

jpantaleoni@nvidia.com

jocohen@nvidia.com